Lecture 1 — The Web, End to End

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 1

Lecturer: Irfan Subakti — yifana@gmail.com

Today no cloud stays a cloud: we follow one page from a key press to a picture on the screen, and then serve a page of our own five ways. Book: Chapter 1, Sections 1.1–1.20.

What This Meeting Covers

1. **The plumbing** — the internet against the web; IP addresses and ports; DNS names; TCP, UDP and QUIC.
2. **The conversation** — one page traced in twenty steps; URLs taken apart; HTTP requests, responses, methods, status codes and headers, down to the byte.
3. **The two ends** — what a web server does with every request; static and dynamic pages; server-side and client-side code; the browser's rendering pipeline.
4. **The big picture** — tiers, the markup lineage, the generations of the web.
5. **Hands on** — the toolchain in VS Code, then one Profitina Laundry page served as a file and from PHP, Java, Python and C#.

The order is the order of dependence: addressing, then naming, then conversation, then what the two programs at the ends do with it. Book: Sections 1.1 and 1.3.

Where We Begin: One Sentence

> **THE ONE SENTENCE.** A program on your machine sends a short piece of text across a wire to a program on another machine, and that other machine sends a longer piece of text back.

HTML, CSS, JavaScript, PHP, MySQL, XML, JSP, ASP.NET, ADO.NET: all of them exist to decide what those two pieces of text say.

The running case is **Profitina Laundry**: three outlets in Surabaya — Rungkut, Keputih and Wiyung — and a finished application with about 3,000 bills of sample data.

The programs live in `WebPro/Code_new/Chapter01/`, one folder per technology: `html`, `php`, `java`, `python`, `aspnet`.

Keep a terminal open and type the commands rather than pasting them; when something works, break it once on purpose and read the failure. Book: Section 1.1.

The Internet Is a Network of Networks

> **DEFINITION.** A **computer network** is two or more computers joined so that one can send data to the other. The **internet** is a worldwide network formed by joining many smaller networks, all agreeing to speak the same family of protocols, **TCP/IP**.

LAN — one building or campus; **WAN** — a city or a country; **wireless LAN** — what the router's sticker calls Wi-Fi.

Nobody built the internet as one network: it is an **agreement** between networks that other people built.

A **protocol** is a set of rules: what to send, in what order, and what to do with what comes back.

A switch in the wall only passes a message from the socket it arrived on to the right one of the others. Book: Sections 1.2.1 and 1.2.2.

Many Clouds, One Agreement

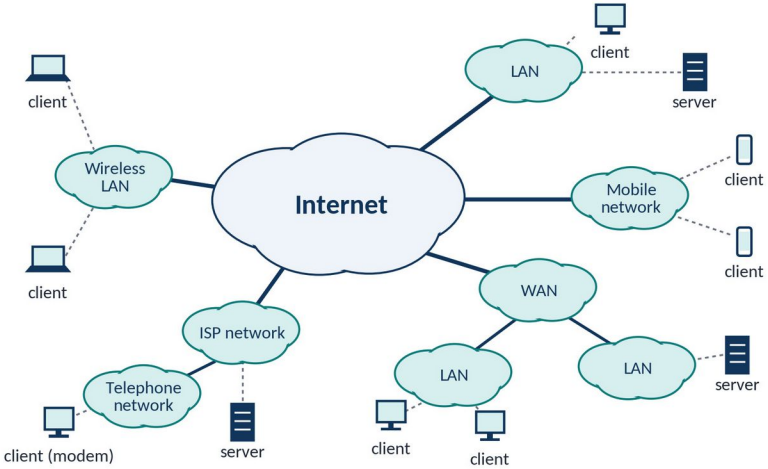


Figure 1.1 The internet as a network of networks: clients and servers sit on LANs, wireless LANs, WANs, mobile networks, an ISP's network and even a telephone network, and all of those join the same middle.

In Figure 1.1 each small cloud is somebody's network — a LAN, a WAN, a mobile network, an ISP. The big cloud in the middle is only the agreement between them. Book: Section 1.2.2.

The Web Is One Service on It

DEFINITION. The **web** — the World Wide Web — is one internet service: the one that runs on the application protocol **HTTP**.

The internet also carries mail (SMTP), file transfer (FTP), games, telephony, streaming — on the internet, not on the web.

Whatever speaks HTTP is part of the web, whether or not a human ever sees the exchange.

KEY INSIGHT. The internet is the road. The web is one company's trucks. Email is a different company's trucks. They share the road; they are not the same business.

In a web application we never write the program that listens on the network: the web server does that difficult work and hands our code a request that is already parsed. Book: Sections 1.2.3–1.2.5.

IPv4: Thirty-Two Bits

> **DEFINITION.** An **IP address** identifies one network interface. **IP** puts data in a labeled envelope, a **packet** (header plus payload), and forwards it toward the address on the label.

IPv4 writes 32 bits as four **octets** in decimal: `103.94.189.4`. Each octet is 0 to 255, because $2^8 = 256$.

Equation (1.1) counts every IPv4 address — about 4.3 billion, fewer than the connected devices on earth.

$$2^{32} = 4\,294\,967\,296 \tag{1.1}$$

> **WORKED EXAMPLE.** $189 = 128 + 32 + 16 + 8 + 4 + 1$, so the octet is 10111101. All of 103.94.189.4 is 01100111 01011110 10111101 00000100 — the dots are for people, and no dot is ever transmitted.

An address with a 256 *in it is not merely wrong, it is unrepresentable. Check each group by adding back its place values:* $64 + 32 + 4 + 2 + 1 = 103$. *Book: Sections 1.3.1 and 1.3.2.*

IPv6, and Ports

IPv6 uses 128 bits: eight groups of four hexadecimal digits, one zero run shortened to `::`, as in `2606:4700:3031::6815:33af`.

$$2^{128} \approx 3.4 \times 10^{38}, \tag{1.2}$$

Equation (1.2) is 2^{96} times Equation (1.1). In 2024 about 28.5 percent of Cloudflare's dual-stack requests already arrived over IPv6.

> **DEFINITION.** A **port** is a sixteen-bit number (0 to 65535) that identifies one listening program on a machine. The pair (IP address, port) is one endpoint of a conversation.

80 HTTP, 443 HTTPS, 3306 MySQL; this course's servers: 8000 static, 8100 PHP, 8080 Java, 5000 Flask, 5100 ASP.NET Core, 5500 Live Server.

Browsers assume port 80 for `http` and 443 for `https`; any other port must be typed. Book: Sections 1.3.3 and 1.3.4.

What IP Does Not Promise

A packet may be **lost**, and never arrive.

Two packets may arrive in the **wrong order**, because they took different routes.

A packet may arrive **twice**, or arrive **damaged**.

> **BEST-EFFORT DELIVERY.** By itself IP is useless for a web page: a page with a missing middle or shuffled paragraphs is not a page. Something must turn it into a reliable, ordered stream — TCP.

IP makes a reasonable effort to forward the packet, and that is its entire commitment. Book: Section 1.3.5.

DNS: Names Read Right to Left

> **DEFINITION.** The **Domain Name System** (DNS) is a worldwide, distributed directory that maps human-readable names onto IP addresses.

```
www . its . ac . id
 ^     ^     ^    ^
 |     |     |    +-- top-level domain: Indonesia
 |     |     +------- second level: academic institutions in Indonesia
 |     +------------- third level: Institut Teknologi Sepuluh Nopember
 +------------------- the name of one machine, or one service, inside ITS
```

Generic TLDs (`.com`, `.org`, `.dev`) and country codes (`.id`, `.jp`); an invisible root, a bare dot, sits right of `id`.

`.ac.id` is restricted to accredited Indonesian academic institutions: a ccTLD usually carries rules about who may hold it. Book: Sections 1.4 and 1.4.1.

The Lookup, Step by Step

1. Your computer asks its **resolver** (your provider, your campus, `8.8.8.8`, `1.1.1.1`): “the address of `www.its.ac.id`?”

2. The resolver asks a **root server**, which answers “ask the servers for `id`”.

3–4. An `id` server says “ask `ac.id`”; an `ac.id` server says “ask `its.ac.id`”.

5–6. The `its.ac.id` server holds the official record and replies `103.94.189.4`; the resolver passes it on, and remembers it.

Every answer carries a **time to live**. Before moving a site, lower it a day ahead, and raise it again afterward.

Four questions to four different organizations, usually in well under a tenth of a second; that is the payoff of distributed. Book: Sections 1.4.2 and 1.4.3.

A Lookup of Our Own

Listing 1.1 calls `socket.getaddrinfo`, the operating system's own lookup — the one every browser ultimately calls.

Listing 1.1 `python/dnslook.py` — a name-to-address lookup in plain Python, with no libraries to install.

```python
# dnslook.py - ask the resolver for the IP addresses behind a name.
import socket
import sys

name = sys.argv[1] if len(sys.argv) > 1 else "www.its.ac.id"

for family, _, _, _, address in socket.getaddrinfo(name, 80, proto=socket.IPPROTO_TCP):
    kind = "IPv4" if family == socket.AF_INET else "IPv6"
    print(f"{name} -> {address[0]:<40} ({kind})")
```

```
profitina.com -> 104.21.51.175                            (IPv4)
profitina.com -> 172.67.183.54                            (IPv4)
profitina.com -> 2606:4700:3031::6815:33af                (IPv6)
profitina.com -> 2606:4700:3037::ac43:b736                (IPv6)
```

Four addresses for one name, two IPv4 and two IPv6: the simplest load balancing, and the way the world moves from 4 to 6 without a flag day. Book: Section 1.4.4.

TCP: a Reliable, Ordered Stream

> **DEFINITION. TCP**, the Transmission Control Protocol, turns IP's unreliable packet delivery into a **reliable, ordered, two-way stream of bytes** between two endpoints.

Sequence numbers put bytes back in place; **acknowledgments** (ACK) say "I have everything up to byte 5,400".

Retransmission on timeout resends what was not acknowledged; **checksums** discard damaged packets.

Flow control protects a slow receiver; **congestion control** protects the shared links.

From the program's side there are no packets at all: bytes go in at one end and come out at the other, like water in a hose. Book: Sections 1.5.1 and 1.5.2.

The Three-Way Handshake

SYN — the client proposes its starting sequence number; **SYN-ACK** — the server acknowledges it and proposes its own; **ACK** — the client acknowledges the server's.

That costs one **round trip** r before any useful byte moves, so over a fresh, unencrypted connection Equation (1.3) holds.

$$t_{\text{first byte}} \geq r_{\text{TCP}} + r_{\text{HTTP}} = 2r\,, \tag{1.3}$$

> **EXAMPLE.** From Surabaya, r to a nearby server may be 10 ms and to Frankfurt 250 ms: a floor of 20 ms against half a second, however fast either server is.

The conversation ends with FIN, or abruptly with RST when something has gone wrong. Book: Section 1.5.3.

TCP, UDP, and QUIC

Table 1.1 TCP and UDP compared. The web chose TCP because a page with a hole in it is not a page.

	TCP	UDP
Connection	Established first, with a handshake	None; each message stands alone
Lost data	Retransmitted until it arrives	Gone
Order	Guaranteed	Not guaranteed
Overhead	Higher	Very low
Delay if a packet is lost	Everything behind it waits for the retransmission	Nothing waits
Good for	Web pages over HTTP/1.1 and HTTP/2, files, mail — anything where every byte matters	Live voice and video, games, DNS queries — and QUIC, which builds its own reliability on top

Table 1.1: TCP's "everything behind a lost packet waits" is head-of-line blocking. QUIC rebuilds TCP's promises per stream on top of UDP, with encryption built in, and HTTP/3 is HTTP over QUIC. Book: Section 1.5.4.

Four Layers, Nested Envelopes

Table 1.2 The four layers of the TCP/IP model. Each layer uses the one below it and knows nothing of the one above.

Layer	Question it answers	Examples	Unit of data
Application	*What do the two programs want to say?*	HTTP, SMTP, FTP, DNS	Message
Transport	*Which program, and did it all arrive?*	TCP, UDP, QUIC	Segment / datagram
Internet	*Which machine, and which way do I forward this?*	IP, ICMP	Packet
Link	*How do I push bits onto this particular wire?*	Ethernet, Wi-Fi	Frame

Read Table 1.2 downward for sending and upward for receiving. Because HTTP does not know about Wi-Fi and Ethernet does not know about pages, either can be replaced without disturbing the other. Book: Section 1.5.5.

The Exchange in One Picture

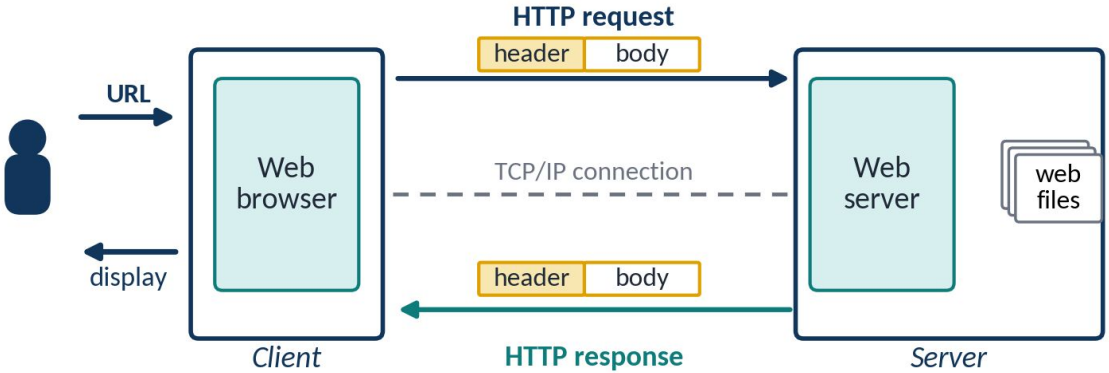


Figure 1.2 The exchange in one picture: URL in, HTTP request out, a TCP/IP connection in between, HTTP response back, display. Each message is a header followed by a body.

Figure 1.2: URL in, HTTP request out, TCP/IP in between, HTTP response back, display. For a page made of several files, the browser sends one more request per file. Book: Section 1.6.

One Page, Traced: Steps 1–10

Table 1.3 One page traced from keystroke to pixel, steps 1 to 10: from the typed name to bytes handed to IP.

#	What happens	Who does it
1	You type `www.its.ac.id` and press Enter	You
2	The browser parses the text into a URL: scheme `https`, host `www.its.ac.id`, path `/`	Browser
3	The browser checks its own cache. If it already has a fresh copy, it stops here and shows it	Browser
4	The browser asks the operating system to resolve the name	Browser
5	The resolver walks root, then `id`, then `ac.id`, then `its.ac.id`, and returns `103.94.189.4`	DNS
6	The browser opens a TCP connection to `103.94.189.4` on port 443	TCP
7	SYN, SYN-ACK, ACK — one round trip — and the connection is established	TCP
8	For `https`, a TLS 1.3 handshake follows: the certificate is checked and keys agreed, in one more round trip	TLS
9	The browser writes the HTTP request bytes into the connection	HTTP
10	TCP chops those bytes into segments, numbers them, and hands them to IP	TCP

Table 1.3: steps 4 and 5 are DNS, 6 and 7 the TCP handshake, 8 the TLS handshake of `https`. Book: Section 1.6.

One Page, Traced: Steps 11–20

Table 1.4 The same trace, steps 11 to 20: across the internet, through the server, and back to the screen.

#	What happens	Who does it
11	IP puts each segment in a packet and forwards it hop by hop across the internet	IP
12	At the far end IP delivers the packets; TCP reorders them and reassembles the stream	TCP
13	The web server reads the request line, the headers, and the body if there is one	Web server
14	The server decides: a file on disk, or a program to run?	Web server
15	If a program, it runs — PHP, a servlet, a Flask view, an ASP.NET page — and produces HTML	Your code
16	The server writes the status line, the response headers, a blank line, and the body	Web server
17	Those bytes travel back the same way, through TCP and IP	TCP/IP
18	The browser parses the HTML and finds references to CSS files, images, scripts	Browser
19	**For each one it sends another HTTP request** — steps 3 to 17 again, often in parallel on the same connection	Browser
20	The browser builds the page and paints it. You see it	Browser

Table 1.4: step 19 is why pages are slow — a modest page is thirty or forty requests. Step 14 is the fork in the road: a file found, or a page computed. Book: Section 1.6.

A URL Has Seven Pieces

`https://sari:rahasia@www.its.ac.id:8443/course/webpro/ch01.html?lang=en&page=3#section-1-7` — the port is used up by the connection, and the **fragment is never sent at all**.

Table 1.5 The seven pieces of a URL, and which of them the server ever gets to see.

Piece	In the example	What it means	Who ever sees it
Scheme	`https`	Which protocol to speak	Browser only
Userinfo	`sari:rahasia`	Credentials. **Deprecated; never use it**	Browser only
Host	`www.its.ac.id`	Which machine, by name or by address	Sent in the `Host` header
Port	`8443`	Which program on that machine. Omitted means 80 for `http`, 443 for `https`	Browser only, except as part of `Host`
Path	`/course/webpro/ch01.html`	Which resource on that server	Sent in the request line
Query	`lang=en&page=3`	Extra parameters, as `name=value` pairs joined by `&`	Sent in the request line
Fragment	`section-1-7`	Which part of the document to scroll to	**Never sent to the server at all**

Table 1.5: no PHP page, servlet, Flask view or ASP.NET page can see the fragment; what the server must know belongs in the path or the query. Book: Sections 1.7 and 1.7.1.

Taking a URL Apart in Python

Listing 1.2 `python/urlparts.py` — a URL split into its pieces by the Python standard library.

```python
# urlparts.py - take a URL apart into the pieces RFC 3986 defines.
from urllib.parse import parse_qs, urlsplit

url = ("https://sari:rahasia@www.its.ac.id:8443"
       "/course/webpro/ch01.html?lang=en&page=3#section-1-7")

parts = urlsplit(url)
print("scheme   :", parts.scheme)
print("username :", parts.username)
print("password :", parts.password)
print("host     :", parts.hostname)
print("port     :", parts.port)
print("path     :", parts.path)
print("query    :", parts.query)
print("parsed   :", parse_qs(parts.query))
print("fragment :", parts.fragment)
```

```
query    : lang=en&page=3
parsed   : {'lang': ['en'], 'page': ['3']}
fragment : section-1-7
```

Listing 1.2: the query became a dictionary of lists, because `?tag=a&tag=b` is legal. PHP's `parse_str` and a simple Java loop keep one value per name and silently lose the first `tag`. Book: Section 1.7.2.

Percent-Encoding

A space, `&`, `?`, `#`, `/` or `%` meant as data must be hidden: a byte b becomes a percent sign and two hexadecimal digits, Equation (1.4).

$$\%\,h_1 h_0 \text{ where } b = 16\,h_1 + h_0,\ 0 \le h_0, h_1 \le 15, \tag{1.4}$$

EXAMPLE. Space is $32 = 16 \cdot 2 + 0$, so `%20`; `&` is $38 = 16 \cdot 2 + 6$, so `%26`; `/` is $47 = 16 \cdot 2 + 15$, so `%2F`.

Forms use an older dialect in which a space is `+`: `%20` anywhere in a URL, `+` inside form-encoded data.

Left raw, the `&` of "cuci kiloan & setrika" starts a second parameter and cuts the value in half. Book: Section 1.7.3.

PHP's Two Encoders

Listing 1.3 `php/encode.php` — PHP's two encoders: `rawurlencode` for URLs, `urlencode` for form data.

```php
<?php
// encode.php - percent-encode a value before it goes into a URL.
$s = 'cuci kiloan & setrika/express';
echo "raw    : $s\n";
echo "encoded: " . rawurlencode($s) . "\n";        // RFC 3986: space -> %20
echo "form   : " . urlencode($s) . "\n";           // HTML forms: space -> +
echo "back   : " . rawurldecode(rawurlencode($s)) . "\n";
```

```
raw    : cuci kiloan & setrika/express
encoded: cuci%20kiloan%20%26%20setrika%2Fexpress
form   : cuci+kiloan+%26+setrika%2Fexpress
back   : cuci kiloan & setrika/express
```

> **A TRAP.** Never glue a URL together from a user's text. Use `quote` (Python), `rawurlencode` (PHP), `URLEncoder.encode` (Java) or `Uri.EscapeDataString` (C#).

Listing 1.3: `rawurlencode` follows RFC 3986, `urlencode` the form rules. Java's `URLEncoder` speaks only the form dialect, so the Java version replaces `+` by `%20` by hand. Book: Section 1.7.3.

An HTTP Request, Complete

Four parts: a **request line** (method, target, version); **header lines** `Name: value`; a **blank line**; an optional **body**.

```
GET /health HTTP/1.1
Host: localhost:8080
User-Agent: rawhttp/1.0
Connection: close
```

Read aloud: *with GET, fetch `/health`, in HTTP 1.1; the site is `localhost:8080`; I am rawhttp 1.0; close when done.*

HTTP is a request–response protocol: one message from the client, one message back, and that is the whole exchange. Book: Sections 1.8 and 1.8.1.

The Methods: Safe and Idempotent

Table 1.6 The HTTP methods. *Safe* means it changes nothing on the server; *idempotent* means doing it twice has the same effect as doing it once.

Method	Means	Has a body?	Safe?	Idempotent?
`GET`	Give me this resource	No	Yes	Yes
`HEAD`	Give me only the headers you would send for `GET`	No	Yes	Yes
`POST`	Here is some data; do something with it	Yes	No	No
`PUT`	Store this data at exactly this address	Yes	No	Yes
`DELETE`	Remove the resource at this address	No	No	Yes
`PATCH`	Here is a partial change to the resource	Yes	No	No
`OPTIONS`	What methods does this address accept?	No	Yes	Yes
`TRACE`	Echo my request back to me	No	Yes	Yes

Table 1.6: `GET` must never change anything — browsers pre-fetch, caches store, crawlers follow every link. The rule for these first chapters: ***`GET` to read, `POST` to change.*** *Book: Section 1.8.2.*

A Response, and Its Status Code

```
HTTP/1.1 200 OK
Server: HelloWeb/2.0 (Java)
Date: Wed, 23 Sep 2026 09:29:03 GMT
Content-type: text/plain; charset=utf-8
Content-length: 3

ok
```

Table 1.7 The five classes of HTTP status code. The first digit is the whole story.

Range	Name	Meaning in plain words
1xx	Informational	Hold on, I am still working
2xx	Success	It worked
3xx	Redirection	It is somewhere else; go and look there
4xx	Client error	**You** did something wrong
5xx	Server error	**I** did something wrong

Table 1.7: the first digit is the whole story; 4xx means you did something wrong, 5xx means I did. The reason phrase is for humans; software reads the number. Book: Sections 1.8.3 and 1.8.4.

The Codes You Will Meet

Table 1.8 The status codes met most often in the first chapters, with the situation that produces each.

Code	Reason phrase	When you see it
`200`	OK	The ordinary answer to a successful `GET`
`201`	Created	A `POST` made something new
`204`	No Content	It worked, and there is deliberately nothing to send back
`301`	Moved Permanently	The address changed forever; update your links
`304`	Not Modified	Your cached copy is still good; here is no body at all
`400`	Bad Request	The request was malformed
`401`	Unauthorized	You have not identified yourself
`403`	Forbidden	You have, and you still may not
`404`	Not Found	No such resource
`500`	Internal Server Error	Your program crashed

> **401 IS NOT 403.** `401`: I do not know who you are. `403`: I know exactly who you are, and the answer is still no.

Table 1.8 is a selection; the book's table adds `302`, `429`, `502` and `503`. Book: Section 1.8.4.

Headers: the Adjectives

One line each: a name, a colon, a space, a value. Names are case-insensitive: `Content-Type` and `Content-type` are one header.

Request: `Host` (required — one IP, many sites), `User-Agent`, `Accept`, `Accept-Language`, `Cookie`, `If-Modified-Since`, `Content-Length`.

Response: `Content-Type`, `Content-Length`, `Date`, `Server`, `Last-Modified`, `Location` (with `3xx`), `Set-Cookie`, `Cache-Control`, `Allow` (with `405`).

> **THE MOST IMPORTANT HEADER.** A body carries no label of its own. Send HTML as `text/plain` and the visitor sees your tags. When output looks like gibberish, look at `Content-Type` first.

The book lists every header with an example value in two tables, one for requests and one for responses. Book: Section 1.8.5.

Speaking HTTP from a Program

Listing 1.4 From `python/rawhttp.py`: one TCP connection, one hand-written HTTP request, one response.

```
REQUEST = (
    "GET /health HTTP/1.1\r\n"
    "Host: localhost:8080\r\n"
    "User-Agent: rawhttp/1.0\r\n"
    "Connection: close\r\n"
    "\r\n"
)

with socket.create_connection(("localhost", 8080), timeout=5) as tcp:
    print("--- connected to", tcp.getpeername())
    tcp.sendall(REQUEST.encode("ascii"))     # the request, as bytes
    reply = b""
    while True:
        chunk = tcp.recv(4096)
        if not chunk:                        # the server closed the connection
            break
        reply += chunk
```

Listing 1.4: `socket.create_connection` performs the whole three-way handshake. The same program in PHP uses `fsockopen`, in Java a `Socket`; all three got 148 bytes back. Book: Section 1.8.6.

Worked Example: 65 Bytes

`od -c` shows the request as the server receives it; every line ends with `\r\n`, and the blank line is one more `\r\n`.

```
0000000   G   E   T       /   h   e   a   l   t   h       H   T   T   P
0000020   /   1   .   1  \r  \n   H   o   s   t   :       l   o   c   a
0000040   l   h   o   s   t   :   8   0   8   0  \r  \n   C   o   n   n
0000060   e   c   t   i   o   n   :       c   l   o   s   e  \r  \n  \r
0000100  \n
0000101
```

Count it: request line 20 characters, `Host` line 20, `Connection` line 17, each plus 2, and the blank line 2 — Equation (1.5).

$$(20+2)+(20+2)+(17+2)+2=65. \tag{1.5}$$

The left-hand numbers are octal, so `0000101` is byte 65, which agrees with `wc -c`. Leave out the last `\r\n` and the server waits forever for a blank line. Book: Section 1.8.7.

Stateless, and Versioned

KEY INSIGHT. HTTP is **stateless**: each request is complete in itself, and the server may forget everything the moment it has answered. That is why any of a thousand servers can answer any request.

HTTP/1.1 (1997) reuses connections and requires `Host`; HTTP/2 (2015) frames the same messages in binary; HTTP/3 (2022) carries them over QUIC. **HTTPS** is HTTP inside TLS 1.3, which costs one round trip more, Equation (1.6).

$$t_{\text{first byte}} \geq r_{\text{TCP}} + r_{\text{TLS 1.3}} + r_{\text{HTTP}} = 3r\,, \tag{1.6}$$

WORKED EXAMPLE. With $r = 40$ ms: `http` needs at least $2r = 80$ ms by Equation (1.3); `https` over TLS 1.3 and TCP at least $3r = 120$ ms.

"Logged in" is rebuilt on top with cookies: `Set-Cookie` in a response, `Cookie` in every later request. The lock means encrypted, and the certificate matched the name — not that the site is honest. Book: Sections 1.8.8 and 1.8.9.

Three Versions on the Layers

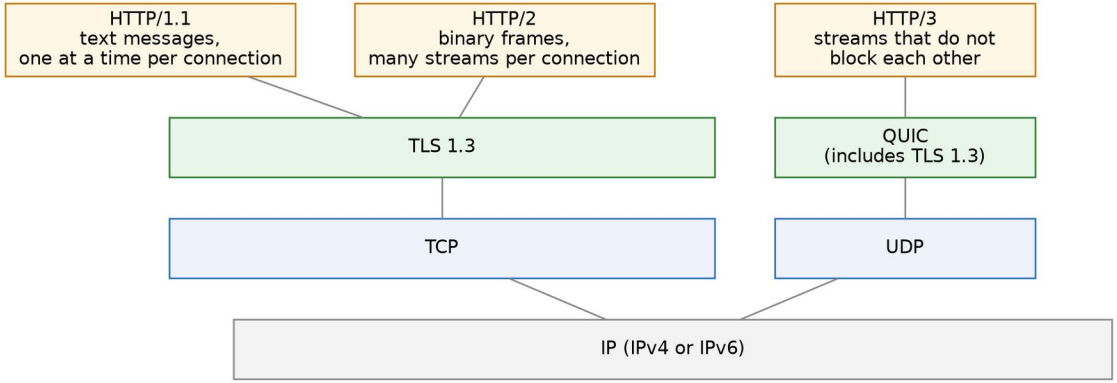


Figure 1.3 HTTP/1.1 and HTTP/2 run over TLS and TCP; HTTP/3 runs over QUIC, which carries TLS 1.3 inside it and runs over UDP. All three carry the same messages.

Figure 1.3: QUIC merges the transport and TLS handshakes, bringing HTTP/3 back to $2r$ *on a first visit. Book: Section 1.8.9.*

A Web Server Is Software

> **DEFINITION.** A **web server** is *software*. It runs on a server machine, and it makes the web documents stored there reachable by users over the internet.

Production: **Apache** (inside XAMPP), **nginx**, **IIS**, **Kestrel** (ASP.NET Core), **Tomcat** (servlets and JSP).

Development only: `php -S`, `python -m http.server`, Flask's server, the JDK's `HttpServer`.

A web server does not "make a website": it makes files and programs *reachable*. What they say is our business.

One machine can run a web server, a database server and a mail server at once; a laptop running today's programs hosts five web servers. Book: Section 1.9.

Nine Things, Every Request

1–3. Accept the connection; read the request line (malformed: `400`); read the headers up to the blank line, and exactly `Content-Length` bytes of body.

4. Choose the site from the `Host` header — **virtual hosting**.

5–6. Map the path into the **document root**, looking for an **index file**; check that the result is still inside the root (`403`, `404`).

7–9. File or program? Choose the `Content-Type`; write status line, headers, blank line, body.

> **A TRAP.** Glue the path onto the root naively and `/../../../etc/passwd` reads a private file: **path traversal**.

Step 7 is step 14 of the trace, the fork in the road. For a file, the `Content-Type` comes from the extension (`.html` is `text/html`, unknown is `application/octet-stream`). Book: Section 1.9.1.

Static and Dynamic Pages

> **DEFINITION.** A **static page** is a file on the server's disk, sent to every visitor unchanged. A **dynamic page** does not exist until it is asked for: a program builds it at the moment of the request.

Reload the static page ten times: nothing changes. Reload the PHP board: *Page built by the server at …* changes every time.

On the wire: a file has `Last-Modified`, so the browser may ask `If-Modified-Since` and get `304 Not Modified` with **no body**.

```
HTTP/1.0 304 Not Modified
Server: SimpleHTTP/0.6 Python/3.11.15
Date: Wed, 23 Sep 2026 09:28:49 GMT
```

Sites with many static files get much of their speed from this one exchange. Book: Sections 1.10.1 and 1.10.2.

Choosing Between Them

Table 1.9 Static and dynamic pages compared. Most real sites are both at once.

	Static	Dynamic
Where the page comes from	A file on disk	A program, run per request
Same for every visitor?	Yes	Not necessarily
Cost per request	A disk read, often from memory	Running a program, usually a database query too
Caching	Easy, and very effective	Hard, and often wrong
Can show live data	No	Yes
Can show data per user	No	Yes
What can break	Almost nothing	Your code, the database, the template, the third-party call
Good for	Documentation, a price list, an article, images, CSS, JavaScript	A dashboard, a basket, a search result, anything signed in

Table 1.9: static is not primitive and dynamic is not grown-up. The finished Laundry application serves its logo, stylesheet and scripts as files and its bill board as a program. Book: Section 1.10.3.

Server-Side and Client-Side

DEFINITION. Server-side code runs on the server machine, before the response is sent. **Client-side** code runs in the visitor's browser, after the response has arrived.

1. **CGI** — a separate program whose output *is* the reply.
2. **Server-side scripting** — PHP, ASP, JSP, Python: code inside the document or behind it; the visitor sees only the output.
3. **Client-side scripting** — JavaScript (ECMAScript), run by the browser without asking the server.
4. **Plug-ins** — applets, ActiveX, Flash: gone, but still met in old code.

The four kinds are organized by who runs the code, which is exactly the right division. Book: Sections 1.11 and 1.11.1.

What Each Side Can See

Table 1.10 What server-side code and client-side code can each see and do. The last three rows are security rules, not trivia.

	Server-side	Client-side
Runs on	The server machine	The visitor's machine
Runs when	Before the response is sent	After the response arrives
Languages here	PHP, Java/JSP, C#/ASP.NET, Python	JavaScript
Can read the database	Yes	**No**, not directly
Can read files on the server	Yes	No
Can hold secrets (passwords, keys)	Yes	**No. Never.**
Can see what the user typed without a request	No	Yes
Can change the page without a request	No	Yes
The user can read the source	No	**Yes, always**
The user can change it	No	**Yes, always**

Table 1.10: the last three rows are security rules. Client-side code is a convenience, never a control: validate on the client for kindness, and again on the server for truth. Book: Sections 1.11.2 and 1.11.3.

Browsers Today

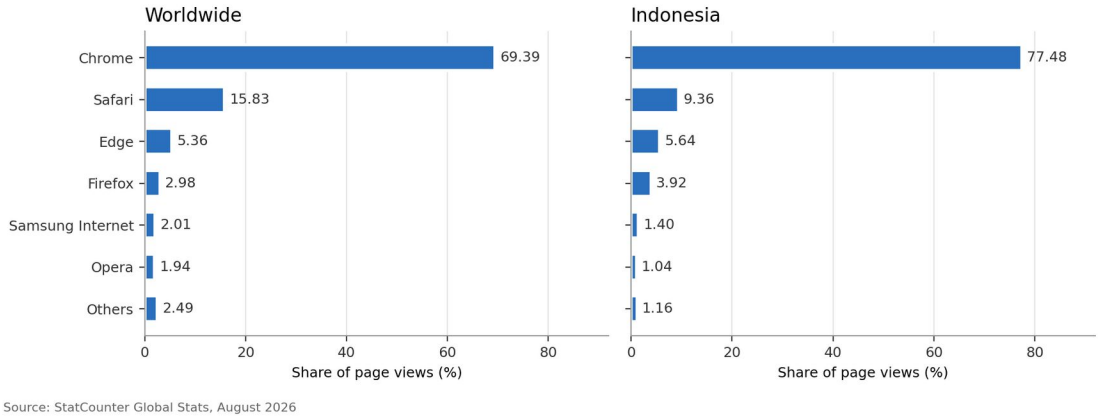


Figure 1.4 Browser share of page views in August 2026, worldwide and in Indonesia. Chrome leads everywhere, and by more in Indonesia, where most browsing happens on Android phones.

Figure 1.4: in August 2026 Chrome held 69.39 percent of page views worldwide and 77.48 percent in Indonesia. All browsers descend from three engines — Blink, WebKit, Gecko. Measure your own visitors; never design from a global pie chart. Book: Sections 1.12.1 and 1.12.2.

The Rendering Pipeline

1. Parse HTML into the DOM — a tree; a missing closing tag is *repaired*, perhaps not as you meant.

2. Fetch every `<link>`, `<img>` and `<script>`; **3. parse CSS** into the CSSOM.

4. Render tree — what will be drawn, with computed style (`display: none` is in the DOM, not in the render tree).

5. Layout (reflow) — the expensive geometry; **6. paint**; **7. composite**.

> **TWO CONSEQUENCES.** An ordinary `<script>` stops parsing, so scripts go at the bottom or get `defer`. Changing a width forces layout again; changing only a color repaints.

The bytes of a page with its script in the head and one with the script at the bottom are identical; the experience is not. Book: Section 1.12.3.

Web Architecture in Tiers

Table 1.11 Web architecture by number of tiers. Three is the shape to assume unless told otherwise.

Tiers	Where the logic lives	Where the data lives	Typical of
One	With the interface	With the interface	A desktop program; a static site
Two	On the server, with the data	On the server	PHP plus MySQL on one host. Chapters 3 and 5
Three	Its own tier	Its own tier	Almost every real web application, including the finished Profitina Laundry
n	Several services	Several stores	Large systems; Profitina itself

Table 1.11: presentation (HTML, CSS, JavaScript), application (PHP, servlet, Flask, ASP.NET), data (MySQL, SQLite). Each can be replaced without touching the others; MVC in Chapter 7 refines the same idea. Book: Section 1.13.

The Smallest Web Page



Figure 1.5 A first HTML document and what Chromium makes of it. The `<title>` appears only on the tab, not in the page, and `<b>first</b>` becomes bold.

Figure 1.5: the `<title>` sits in `<head>` and is not in the page; `<br>` has no partner; `<b>first</b>` wraps what it affects, and that wrapping is what makes the document a tree. Book: Sections 1.14.1 and 1.14.2.

Twenty Years on One Chart

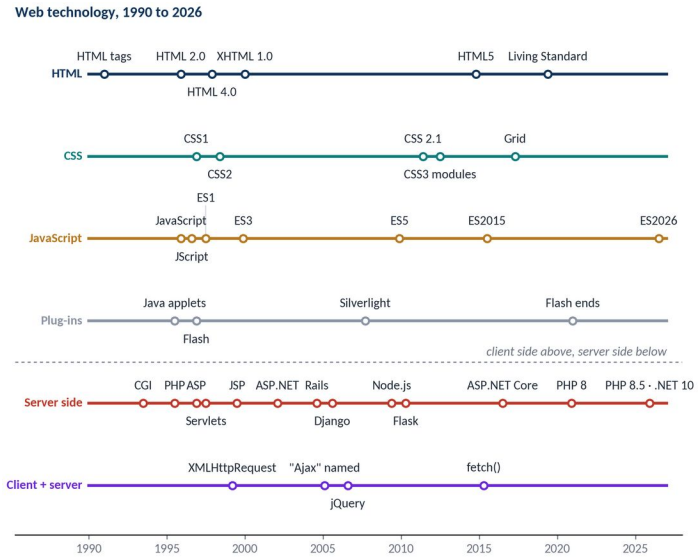


Figure 1.6 Web technology, 1990 to 2026: client-side lanes above the dashed rule, server-side below it, and the client-and-server techniques at the bottom. Every language this book teaches is on it.

Figure 1.6: client side above the dashed rule, server side below, and the client-and-server techniques at the bottom — Section 1.11 drawn across three and a half decades, from the first HTML tags of 1991 to PHP 8.5 and .NET 10 in 2025. The markup line runs Scribe, SGML, HTML, XML, XHTML, and today's HTML Living Standard. Book: Sections 1.14.1 and 1.14.3.

Read, Write, Meaning, Agents

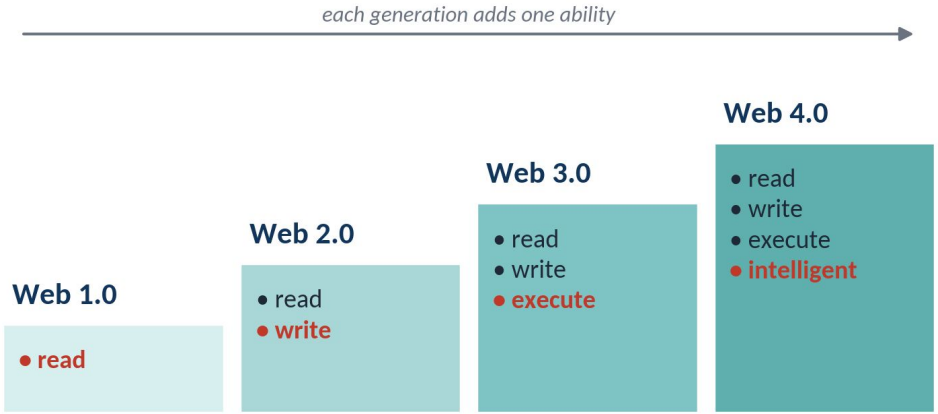


Figure 1.7 The generations in one word each: each adds one ability to the last — read, then write, then execute, then intelligent.

Figure 1.7: Web 1.0 read, Web 2.0 write (users as "prosumers"), Web 3.0 meaning (the Semantic Web), Web 4.0 intelligent agents. Web3 is a different thing: Web 3.0 is about meaning, Web3 about ownership. Book: Section 1.15.

The Toolchain, Once

VS Code with Live Server, PHP Intelephense, Extension Pack for Java, Python and C# Dev Kit. Open a *folder*, and use the built-in terminal.

Browser developer tools (F12): Elements is the DOM, Network is Section 1.8 live, Console, Sources.

PHP 8.2+, current release 8.5 (XAMPP 8.2.12 or a package manager), **MySQL/MariaDB** via XAMPP, **JDK 25**, **Python 3.14 + Flask** in a virtual environment, **.NET 10 SDK**, **Git**.

Verify in a *new* terminal: `php --version`, `javac -version`, `python3 -m flask --version`, `dotnet --version`, `git --version`.

Install kits, not runtimes: `javac` must answer as well as `java`, and the .NET SDK, not only its runtime. Book: Sections 1.16.1–1.16.9.

One Client, Five Servers

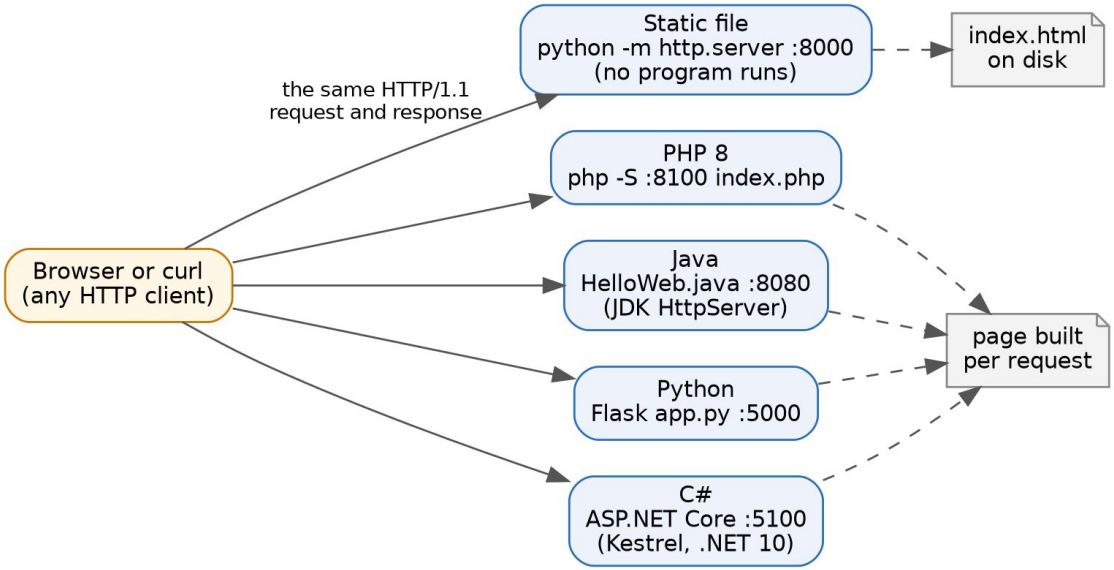


Figure 1.8 One client, five servers. The static server reads a file; the other four build the page for each request. All five speak the same HTTP.

Figure 1.8: the static server reads a file; PHP, Java, Python and C# build the Keputih bill board for each request, with `/`, `/bills/…`, `/health` and `404`. Book: Section 1.17.

Worked Example: the Charge

Every server computes each bill's charge by Equation (1.7).

$$\text{charge} = \text{units} \times \text{price per unit}. \tag{1.7}$$

> **EXAMPLE.** B2943: 5 kg of wash, dry and fold at Rp 8,000 gives $5 \times 8000 = 40\,000$. B2898: 2 shirts of dry cleaning at Rp 20,000 gives 40 000. B2920: 6 kg express at Rp 12,000 gives 72 000.

Total: $40\,000 + 40\,000 + 72\,000 = 152\,000$, printed as `3 bills, Rp 152,000 in total.`

Add bill B2951, one bed cover at Rp 35,000, and the last line becomes `4 bills, Rp 187,000 in total.` — nothing else changes, because the table is computed from the data. Book: Section 1.17 and Exercise 1.18.

Way One: a Plain File

Listing 1.5 The body of `html/index.html` — a static page. Every visitor gets these bytes, unchanged.

```
<body>
  <img src="profitina-logo-horizontal.png" alt="Profitina" width="177" height="51">
  <h1>Profitina Laundry Keputih</h1>
  <p>Jl. Flamboyan No. 8, Keputih, Sukolilo, Surabaya 60111.
     Open 07:00 to 21:00, every day.</p>

  <h2>Price list</h2>
  <table>
    <tr><th>Service</th><th>Price</th></tr>
    <tr><td>Wash, dry &amp;amp; fold</td><td>Rp 8,000 per kg</td></tr>
    <tr><td>Express wash &amp;amp; fold, 1 day</td><td>Rp 12,000 per kg</td></tr>
    <tr><td>Ironing only</td><td>Rp 6,000 per kg</td></tr>
    <tr><td>Dry cleaning, shirt or blouse</td><td>Rp 20,000 per item</td></tr>
  </table>

  <p class="note">This page is a file. It was last edited by hand on 1 September 2026.</p>
</body>
```

Listing 1.5: serve it with Live Server, `python3 -m http.server 8000` or `php -S localhost:8001`. Never double-click it: a `file://` address means no server, no TCP, no HTTP. Book: Section 1.17.1.

Way Two: PHP Decides What to Answer

Listing 1.6 From `php/index.php`: the request's method and path, then the answers for a wrong method and for `/health`.

```
$method = $_SERVER['REQUEST_METHOD'];
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
header('Server: HelloWeb/2.0 (PHP)');

if ($method !== 'GET' && $method !== 'HEAD') {
    http_response_code(405);
    header('Allow: GET, HEAD');
    header('Content-Type: text/plain; charset=utf-8');
    echo "405 Method Not Allowed\n";
    exit;
}

if ($path === '/health') {
    header('Content-Type: text/plain; charset=utf-8');
    echo "ok\n";
    exit;
}
```

Listing 1.6: `header(…)` and `http_response_code(…)` must run before the first byte of body. Naming `index.php` on the command line, `php -S localhost:8100 index.php`, makes it a router script that receives every request. Book: Section 1.17.2.

Way Two: HTML with Holes in It

Listing 1.7 The end of `php/index.php`: the board, HTML copied through with `<?= … ?>` holes filled at the moment of the request.

```
<?php else:
    $total = 0;
    foreach ($bills as $b) {
        $total += $b['units'] * $b['price'];
    } ?>
  <h1>Profitina Laundry Keputih</h1>
  <p>Page built by the server at <?= date('j F Y, H:i:s') ?> WIB.</p>
  <table>
    <tr><th>Bill</th><th>Customer</th><th>Service</th><th>Units</th><th>Status</th><th>Charge</th></tr>
<?php foreach ($bills as $b): ?>
    <tr><td><a href="/bills/<?= e($b['id']) ?>"><?= e($b['id']) ?></a></td><td><?= e($b['customer']) ?
></td><td><?= e($b['service']) ?></td><td><?= $b['units'] ?></td><td><?= e($b['status']) ?></td><td><?=
rupiah($b['units'] * $b['price']) ?></td></tr>
<?php endforeach; ?>
  </table>
  <p><?= count($bills) ?> bills, <?= rupiah($total) ?> in total.</p>
```

Listing 1.7: `e()` wraps `htmlspecialchars`, so the `&` of "Wash, dry & fold" arrives as `&`. PHP sends no `Content-Length`: it closes the connection to mark the end of the body. Book: Section 1.17.2.

Way Three: Java Routes the Request

Listing 1.8 From `route` in `java/HelloWeb.java`: every request arrives here, and one of five answers is chosen.

```
        if (!method.equals("GET") && !method.equals("HEAD")) {
            // The path may exist, but this verb is not allowed on it.
            exchange.getResponseHeaders().add("Allow", "GET, HEAD");
            send(exchange, 405, "text/plain; charset=utf-8", "405 Method Not Allowed\n");
            return;
        }

        if (path.equals("/")) {
            send(exchange, 200, "text/html; charset=utf-8", boardPage());
        } else if (path.equals("/health")) {
            send(exchange, 200, "text/plain; charset=utf-8", "ok\n");
        } else if (path.startsWith("/bills/") && findBill(path.substring(7)) != null) {
            send(exchange, 200, "text/html; charset=utf-8", billPage(findBill(path.substring(7))));
        } else {
            send(exchange, 404, "text/html; charset=utf-8",
                    "<h1>404 Not Found</h1>\n<p>No page lives at " + escape(path) + ".</p>\n");
        }
```

Listing 1.8 is steps 2 and 7 of the nine, written out. The server needs a JDK and nothing else: `javac HelloWeb.java`, then `java HelloWeb 8080`. Book: Section 1.17.3.

Way Three: Writing the Response

Listing 1.9 `send` in `java/HelloWeb.java`: status line, headers, blank line, body — with the true `Content-Length`.

```
    static void send(HttpExchange exchange, int status, String contentType, String body)
            throws IOException {
        byte[] bytes = body.getBytes(StandardCharsets.UTF_8);

        Headers out = exchange.getResponseHeaders();
        out.set("Content-Type", contentType);
        out.set("Server", "HelloWeb/2.0 (Java)");

        // The second argument is Content-Length. Sending the true length tells
        // the browser exactly when the body has finished arriving. A HEAD
        // request gets the same headers and no body (-1 means "no body").
        boolean head = exchange.getRequestMethod().equals("HEAD");
        exchange.sendResponseHeaders(status, head ? -1 : bytes.length);
        if (!head) {
            try (OutputStream stream = exchange.getResponseBody()) {
                stream.write(bytes);
            }
        }
        exchange.close();
    }
```

Listing 1.9: `-1` means "no body", so `HEAD` gets the headers alone. Java answers `Content-length: 1001` where PHP closed the connection; `/nope` gets `404`, and a `POST` gets `405` with `Allow: GET, HEAD`. Book: Section 1.17.3.

Way Four: Python and Flask

Listing 1.10 From `python/app.py`: each `@app.route` maps a URL pattern to a function.

```python
@app.route("/")
def board():
    # Everything below is computed now, at the moment of the request.
    now = datetime.now(WIB).strftime("%d %B %Y, %H:%M:%S WIB").lstrip("0")
    total = sum(b["units"] * b["price"] for b in BILLS)
    return render_template("home.html", now=now, bills=BILLS,
                           total=rupiah(total), rupiah=rupiah)


@app.route("/bills/<bill_id>")
def one_bill(bill_id):
    for bill in BILLS:
        if bill["id"] == bill_id:
            return render_template("bill.html", b=bill, rupiah=rupiah)
    abort(404)                          # Flask turns this into a 404 response


@app.route("/health")
def health():
    return jsonify(status="ok", bills=len(BILLS))
```

Listing 1.10: `/bills/<bill_id>` maps a family of URLs onto one function — step 5 of the nine, done by a pattern instead of a file. The HTML is a Jinja template, which escapes every `{{ … }}` by default. Book: Section 1.17.4.

Way Five: C# and ASP.NET Core

Listing 1.11 From `aspnet/Program.cs`: a minimal API on .NET 10; each `MapGet` does what `@app.route` does in Flask.

```
var app = WebApplication.CreateBuilder(args).Build();
string url = args.Length > 0 ? args[0] : "http://localhost:5100";

// Each MapGet line maps a URL pattern to a function, as @app.route does in Flask.
app.MapGet("/", () => Results.Content(BoardPage(), "text/html; charset=utf-8"));
app.MapGet("/health", () => Results.Text("ok\n", "text/plain; charset=utf-8"));
app.MapGet("/bills/{id}", (string id) =>
{
    Bill? b = bills.FirstOrDefault(x => x.Id == id);
    return b is null ? Results.NotFound() : Results.Content(BillPage(b), "text/html; charset=utf-8");
});

Console.WriteLine($"Profitina Laundry (ASP.NET Core) listening on {url}/");
app.Run(url);
/// <summary>One bill: the same columns as the Bill table of the Laundry database.</summary>
record Bill(string Id, string Customer, string Service, int Units, int Price, string Status)
{
    public int Charge => Units * Price;
}
```

Listing 1.11: `dotnet run` builds and starts it on Kestrel at port 5100. With nothing more asked of it, a missing bill gets `404` and a `POST` gets `405` with `Allow: GET`, both with empty bodies. Book: Section 1.17.5.

Five Ways, One Protocol

Table 1.12 The same page five ways. Everything differs except the last row, and the last row is the point.

	Static file	PHP	Java	Python	C#
What serves it	Any web server	`php -S`, or Apache	JDK `HttpServer`, or Tomcat	Flask (Werkzeug)	Kestrel
Page exists on disk?	Yes	No	No	No	No
Changes between reloads?	No	Yes	Yes	Yes	Yes
`Last-Modified` sent?	Yes (Python's server)	No	No	No	No
`Content-Length` sent?	Yes	No: `Connection: close`	Yes	Yes	Yes
Status codes seen	200, 304, 404	200, 404, 405	200, 404, 405	200, 404, 405	200, 404, 405
`/health` answers	—	`ok`, text	`ok`, text	JSON	`ok`, text
Lines of code	0	about 95	about 165	about 55, plus a 17-line template	about 80
The protocol	**HTTP**	**HTTP/1.1**	**HTTP/1.1**	**HTTP/1.1**	**HTTP/1.1**

Table 1.12: everything differs except the last row, and the last row is the point. Stop every server with Ctrl+C, or the next one on that port fails with address already in use. Book: Section 1.17.6.

Profitina in Practice

The scanner is an HTTP client: resolve, connect, TLS, `GET`, status, headers, body. A site answering `500` cannot be cited by an AI answer engine.

Did the server send the words, or a program that writes them? Text written by client-side JavaScript exists only for clients that run scripts.

`robots.txt`, `llms.txt`, structured data — all are HTTP, and all must be in the response.

Health checks gate releases, as the `/health` routes do today; and UU PDP 27/2022 applies from the first request.

The platform advises; the site owner decides. The case you build with your own keyboard is always Profitina Laundry. Book: Section 1.18.

Common Mistakes

“The internet” for “the web”; a web server taken for a machine; an HTML file opened by double-click; a PHP file opened with Live Server.

Expecting the fragment on the server; building a URL from raw user text.

`GET` that changes something; trusting client-side validation; a secret in JavaScript.

A wrong `Content-Type`; a forgotten blank line; `401` for `403`, or looking at the browser when the log explains the `500`.

A runtime instead of a kit, an old terminal after a PATH change, two servers on one port.

Thirteen mistakes are listed in the book, each with the section that prevents it. Book: Section 1.19.

Summary: Five Things to Remember

1. The internet is a network of networks speaking TCP/IP; the web is the service that speaks HTTP on it.

2. IP addresses machines and promises nothing; DNS turns names into addresses; TCP (or QUIC) makes a reliable, ordered stream.

3. An HTTP message is a first line, headers, a blank line (`\r\n\r\n`) and a body; `4xx` is the client's fault, `5xx` the server's; `Content-Type` decides what the body is.

4. Static pages are files, dynamic pages are programs; server-side code can keep secrets, client-side code can keep none.

5. One page served five ways gave one protocol every time — the subject of the whole course.

The fragment never reaches the server, and data inside a URL is percent-encoded. Book: Section 1.20.

Next Time

Chapter 2 — HTML, CSS, and the Basic Web Languages. The document as a tree, text and lists, links, tables, semantic HTML5, forms, GET and POST on the wire.

CSS from the first rule to the cascade, the box model, flexbox, grid and media queries; JavaScript, the DOM and `fetch`.

Before then: install and verify the toolchain, open the Network panel on a site you use, and work Exercises 1.9, 1.10, 1.11 and 1.16.

The first HTML document of today is where Chapter 2 begins. Book: Sections 1.21 and 1.22.

Lecture 2 — HTML, CSS, and the Basic Web Languages

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 2

Lecturer: Irfan Subakti — yifana@gmail.com

Today we build the public face of Profitina Laundry by hand — a price page, a status page and a pickup form — and learn what every tag, rule and line of script in them does. Book: Chapter 2, Sections 2.1–2.29.

What This Meeting Covers

1. **HTML** — tags, elements and attributes; the document tree; text, lists, links, images, tables; semantic HTML5 and accessibility.
2. **Forms and the wire** — every control, `name` and `<label>`, validation attributes; GET and POST byte for byte; percent-encoding; validation in three places.
3. **CSS** — rules and selectors, color and units, custom properties, the cascade as arithmetic, the box model, flexbox, grid and media queries.
4. **JavaScript** — the language in one sitting, the DOM, events, `fetch` and JSON; structure, presentation and behavior kept apart.
5. **Profitina Laundry** — three public pages, served and checked identically by Java, Python and PHP.

Nothing will be magic: by the end you can press Ctrl+U on any page and read its source. Book: Sections 2.1 and 2.3.

Where We Left Off: a Browser Asks

Chapter 1: **a browser asks, and a server answers**, and the answer is mostly text. Today is about that text.

Static: the file already exists, and the server hands the bytes over unchanged — three steps.

CGI: the server starts a separate program for every request and sends what it prints — five steps.

Embedded scripts: a file of HTML with pockets of code, run by the server — PHP, JSP, ASP.NET — four steps.

> **KEY INSIGHT.** However clever the server, it can only send what the browser understands: HTML, CSS, JavaScript, JSON, images. They are the permanent target of every server-side technology in this course.

The last arrow always points at the browser, and what travels along it is always HTML. Book: Section 2.1.

What a Markup Language Is

> **DEFINITION.** *Markup* is text added to a document that describes the document rather than forming part of it. A **markup language** is an agreed set of rules for writing that description so that a machine can act on it.

HyperText — text with links; **Markup Language** — a description of a document. HTML in three words: **it's about tags**.

HTML has no variables, arithmetic, decisions or loops; the computing starts with JavaScript.

Four jobs: show a document; arrange its content; make forms; include images, audio and video.

Write HTML in a plain-text editor such as VS Code with Live Server, never in Word, which swaps straight quotation marks for curly ones. Save as `.html`, in UTF-8. Book: Sections 2.2 and 2.3.

Tags, Elements, Attributes

> **DEFINITION.** An **element** is a start tag, its content and its end tag, taken together; a **tag** is just the marker. `<p>` is a tag; `<p>Hello</p>` is an element.

```
<input type="text" id="name" name="name" maxlength="60" required>
```

The pattern never changes: `<ELEMENT ATTRIBUTE="VALUE">`. Always quote the value; `required` is a **boolean** attribute.

Void elements have no content and no end tag: `<br>`, `<img>`, `<hr>`, `<input>`, `<meta>`, `<link>`.

Tag names are not case-sensitive; modern practice writes lower case.

`<body bgcolor="blue">`: element `body`, attribute `bgcolor`, value `blue`. Book: Sections 2.4.1 and 2.4.2.

A Document Is a Tree

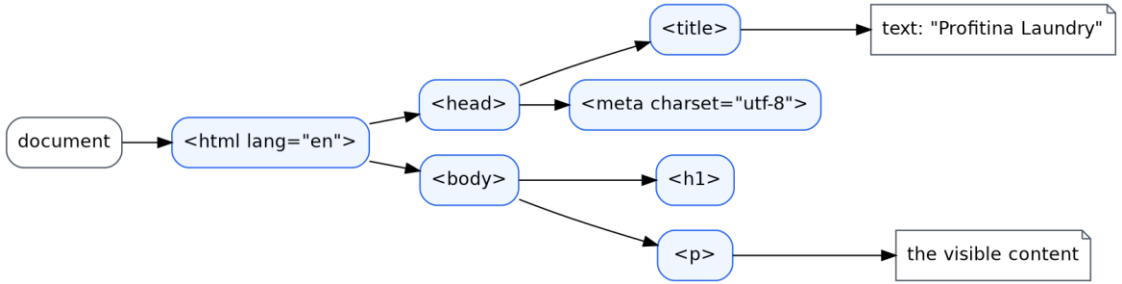


Figure 2.1 An HTML document drawn as a tree. The head carries information about the page; the body carries everything the visitor sees.

***Figure 2.1**: every element but `<html>` has exactly one parent, so tags nest like brackets — `<p>A <b>bold</b> word.</p>`, never `<p>A <b>bold word.</p></b>`. Book: Section 2.4.3.*

The Skeleton of Every Page

Listing 2.1 The skeleton of every HTML page (`html/basics/skeleton.html`). Copy it for every new page.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Profitina Laundry</title>
</head>
<body>
  <h1>Profitina Laundry</h1>
  <p>Three outlets in Surabaya: Rungkut, Keputih and Wiyung.</p>
</body>
</html>
```

`<!DOCTYPE html>` avoids *quirks mode*; `lang` guides screen readers; `charset` stops `é` turning into `Ã©`; the viewport stops a phone pretending to be a monitor.

***Listing 2.1**: the `<title>` labels the tab, names the bookmark and becomes the headline of a search result — the highest-value string in the document. Book: Sections 2.4.4–2.4.7.*

Text: Headings and Paragraphs

> **DEFINITION.** A **block-level** element takes a whole band of the page (paragraphs, headings, lists, tables). An **inline** element sits inside a line of text (bold, italic, links, images).

`<h1>` to `<h6>`: **the number is a rank, not a size.** One `<h1>` per page, never skip a level going down, never choose a level for its look.

HTML collapses whitespace: any run of spaces, tabs and newlines becomes one space. `<br>` ends a line in an address; `<pre>` keeps every space.

Screen readers, search engines and AI answer engines all read the heading outline.

A row of `<br><br><br>` is a beginner's signature: two paragraphs already have space between them. Book: Sections 2.5.1–2.5.4.

Say What You Mean

Table 2.1 Logical formatting tags. They name a meaning, and the appearance follows.

Tag	Meaning	Usual appearance
`<em>` … `</em>`	emphasized	italic, like `<i>`
`<strong>` … `</strong>`	strongly important	bold, like `<b>`
`<del>` … `</del>`	deleted from the document	struck through
`<ins>` … `</ins>`	inserted into the document	underlined, like `<u>`

Physical tags — `<b>`, `<i>`, `<u>`, `<sup>`, `<sub>` — name an appearance; logical tags name a meaning.

> **KEY INSIGHT.** Prefer the logical tag: a voice has no ink, but it can convey importance. `E=MC<sup>2</sup>` and `H<sub>2</sub>O` are the honest uses of physical tags, because the position *is* the meaning.

***Table 2.1**: on screen `<strong>` looks like `<b>`, but one `strong { }` rule can restyle every important word later. Book: Sections 2.5.5 and 2.5.6.*

Lists

Listing 2.2 The body of `html/basics/list-nested.html`: the inner `<ul>` lives inside an `<li>`, never between two of them.

```
<h1>Our services</h1>
<ul>
  <li>By weight
    <ul>
      <li>Wash, dry &amp;amp; fold</li>
      <li>Express wash &amp;amp; fold, 1 day</li>
      <li>Ironing only</li>
    </ul>
  </li>
  <li>Per item or pair
    <ul>
      <li>Dry cleaning</li>
      <li>Shoes deep clean</li>
    </ul>
  </li>
</ul>
```

***Listing 2.2**: `<ul>` when order does not matter, `<ol>` when it does — the browser numbers, so an inserted item never leaves the rest wrong — and `<dl>` with `<dt>` and `<dd>` for name-and-value pairs. Book: Section 2.6.*

Links and Relative Addresses

Table 2.2 Relative addresses. Two dots mean “the folder above”; a leading slash means “start again at the site root”.

Written	Means
`contact.html`	in the same folder as the current page
`pages/contact.html`	in a `pages` folder inside the current folder
`../contact.html`	one folder up, then `contact.html`
`/contact.html`	from the very top of this site, whatever folder we are in

`<a href="#prices">` jumps to `<h2 id="prices">` in the same document.

Link text describes the destination (never *click here*); `tel:+623100002101` becomes tap-to-call; never leave `href` off.

***Table 2.2**: when a link does not work, the file is almost always not where you think it is relative to the page that names it. Book: Section 2.7.*

Images: `alt`, `width`, `height`

Listing 2.3 From `html/basics/image.html`: the official logo with its alternative text and its dimensions.

```
<img src="../assets/profitina-logo-horizontal.png" alt="Profitina" width="236" height="68">
```

> `alt` **IS NOT OPTIONAL.** Write what the picture *conveys* (`alt="Profitina"`), and `alt=""` for pure decoration. Without the attribute a screen reader reads the file name.

`width` and `height` **reserve space**, so the page does not jump when the picture arrives. JPEG for photos, PNG for sharp edges, SVG for icons, WebP and AVIF for smaller photos.

***Listing 2.3**: `src` is an address, relative or absolute, exactly like `href`. Book: Section 2.8.*

HTML Never Complains

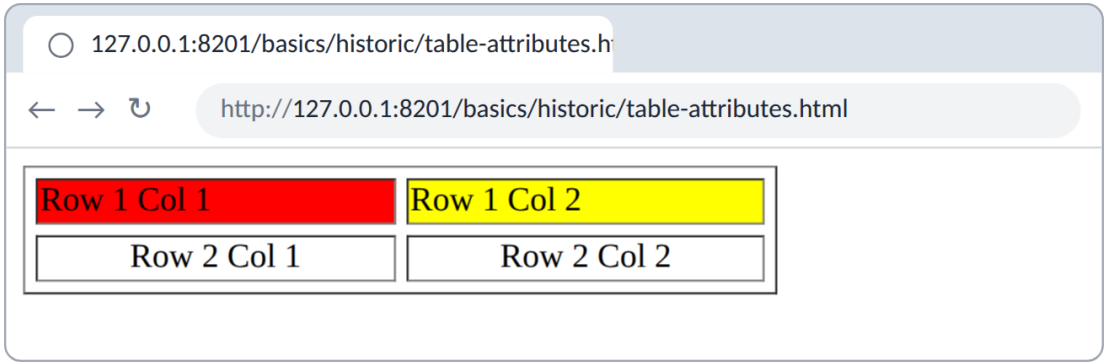


Figure 2.2 The formatted table. The gaps between cells come from `cellspacing`; the padding inside them is missing, because the attribute was misspelled and HTML silently ignores what it does not recognize.

***Figure 2.2**: `cellpading` has one D, so it is silently ignored. The W3C checker (`vnu.jar`) finds it in a second. Book: Sections 2.9.3 and 2.14.1.*

A Table, Written Today

Listing 2.4 A modern table with `<thead>`, `<tbody>`, `<tfoot>`, `<th scope>` and `colspan` (excerpt of `html/basics/price-table.html`).

```
<table class="receipt">
  <caption>Profitina Laundry price list</caption>
  <thead>
    <tr>
      <th scope="col">Service</th>
      <th scope="col">Unit</th>
      <th scope="col">Price (Rp)</th>
    </tr>
  </thead>
  <tbody>
    <tr><th scope="row">Wash, dry &amp;amp; fold</th><td>kilogram</td><td>8.000</td></tr>
    <tr><th scope="row">Express, 1 day</th>   <td>kilogram</td><td>12.000</td></tr>
    <tr><th scope="row">Dry cleaning, shirt</th><td>item</td>  <td>20.000</td></tr>
    <tr><th scope="row">Shoes deep clean</th> <td>pair</td>    <td>45.000</td></tr>
  </tbody>
  <tfoot>
    <tr><td colspan="3">Weight is rounded up to the next whole kilogram.</td></tr>
  </tfoot>
</table>
```

***Listing 2.4**: `<th scope="col">` and `scope="row"` let a screen reader announce `12.000` as the Price of Express, 1 day; `<thead>` repeats on every printed page; `colspan="3"` spans three columns. Tables are for data, never for layout. Book: Section 2.9.4.*

Semantic HTML5

Listing 2.5 The same page in semantic HTML (excerpt of `html/basics/semantic.html`). Every element names what it is.

```
<header>
  <p><a href="index.html">Profitina Laundry</a></p>
  <nav aria-label="Main">
    <ul><li><a href="index.html">Home</a></li></ul>
  </nav>
</header>
<main>
  <h1>Our prices</h1>
  <p>Wash, dry and fold is Rp 8.000 per kilogram.</p>
</main>
<footer>
  <address>Jl. Flamboyan No. 8, Keputih, Surabaya</address>
</footer>
```

The same page built from `<div class="header">`, `<div class="nav">` and `<div class="text">` can look identical and means nothing.

***Listing 2.5**: a screen reader jumps straight to `<main>`, and an answer engine finds the heading, the paragraph and the price. The test: with all the CSS removed, could a stranger still tell what each part is? Book: Section 2.10.*

Accessibility from the First Line

`lang`, one `<h1>`, landmarks, a skip link, `alt`, a `<label>` for every control, a visible focus ring, controls 44 pixels tall — each costs one attribute or one rule.

ARIA's first rule: do not use it when plain HTML already says it. A `<button>` is a button; `<div role="button">` is not, for the keyboard.

WCAG 2.2 asks for text contrast of at least 4.5 to 1, by Equation (2.1).

$$\text{contrast} = \frac{L_{\text{lighter}} + 0.05}{L_{\text{darker}} + 0.05} \tag{2.1}$$

WORKED EXAMPLE. Black on white: $(1 + 0.05)/(0 + 0.05) = 21$. Our hint gray `#4b5563`, $L \approx 0.089$: $1.05/0.139 \approx 7.6$, passes. `#9ca3af`, $L \approx 0.364$: $1.05/0.414 \approx 2.5$, fine for a border, unreadable for text.

Test the way it is used: put the mouse away and use Tab, Enter and Space; zoom to 200 percent; listen with Narrator, VoiceOver or Orca. Book: Section 2.11.

Forms: `action`, `method`, `name`

```
<form action="pickup" method="post">
  <input type="text" name="phone">
  <button type="submit">Send</button>
</form>
```

`action` — where to send; `method` — `get` or `post`; `enctype` — how to package (files need `multipart/form-data`).

`name` **is the whole point:** a control without it is simply not sent, with no error. `id` is for CSS, JavaScript and `<label for>`.

`<label for>` names the control's `id`: a bigger tap target and a spoken name. A `placeholder` is a hint, never a label.

A form inside another form is silently discarded; `<fieldset>` and `<legend>` give a group of controls a name. Book: Sections 2.12.1–2.12.4.

The Input Types That Surprise

Table 2.3 A selection of `<input>` types, the control each produces, and what the server receives.

type	What the visitor sees	What is sent
`text`	a single-line box	whatever was typed
`email`	a text box; a phone shows an @ keyboard	the text, checked for the shape of an address
`date`	a date picker	`YYYY-MM-DD`
`checkbox`	a box that ticks	the `value`, **only when ticked**
`radio`	one of a set of circles	the `value` of the one chosen
`file`	a Browse button	the file's bytes, under `multipart/form-data`
`hidden`	nothing	the `value`, unchanged

***Table 2.3**: an unticked checkbox sends **nothing** — not an empty string; radio buttons group by `name`; a hidden field is invisible, not secret. Book: Section 2.12.5.*

A Grouped Drop-Down List

Listing 2.6 From `html/pickup.html`: `value` is what the server receives; the text between the tags is what the visitor reads.

```
        <select id="service" name="service" required aria-describedby="service-error">
          <option value="">Choose a service</option>
          <optgroup label="By weight">
            <option value="S0001" data-price="8000">Wash, dry &amp;amp; fold - Rp 8.000/kg</option>
            <option value="S0002" data-price="12000">Express wash &amp;amp; fold, 1 day - Rp 12.000/kg</option>
            <option value="S0003" data-price="6000">Ironing only - Rp 6.000/kg</option>
            <option value="S0010" data-price="15000">Curtain - Rp 15.000/kg</option>
          </optgroup>
          <optgroup label="Per item or pair">
            <option value="S0004" data-price="20000">Dry cleaning, shirt/blouse - Rp 20.000</option>
          </optgroup>
        </select>
```

***Listing 2.6**: the server receives `S0001`, the key of the Service table; the first option has `value=""`, so with `required` the visitor must choose; `data-price` is there for scripts. Always write a `<button>`'s `type`. Book: Section 2.12.6.*

The Attributes That Validate

Table 2.4 The validation attributes. The browser applies them before sending, and refuses to send while any of them fails.

Attribute	On	Meaning
`required`	most controls	must not be empty
`minlength`, `maxlength`	text controls	fewest and most characters
`min`, `max`	numbers, dates, times	smallest and largest allowed
`step`	numbers, dates, times	the allowed increment
`pattern`	text controls	a regular expression the value must match
`type`	`email`, `url`, `number`	the type's own built-in check
`readonly`	most controls	cannot be edited, but **is** submitted
`disabled`	most controls	cannot be edited, and is **not** submitted

***Table 2.4**: `readonly` is submitted, `disabled` is not. The phone field uses `pattern="0[0-9 \-]{9,16}"`: a zero, then nine to sixteen digits, spaces or hyphens. Book: Section 2.12.7.*

Consent Is One Line of HTML

Listing 2.7 The last fieldset: an explicit consent checkbox, a hidden field and the two buttons (excerpt of `html/pickup.html`).

```
        <legend>Confirm</legend>
        <label class="choice">
          <input type="checkbox" name="consent" value="yes" required aria-describedby="consent-error">
          I agree that Profitina Laundry stores my name, phone number and address to
          collect and return this laundry. <abbr title="required">*</abbr>
        </label>
        <span class="field-error" id="consent-error"></span>
        <input type="hidden" name="source" value="web">
        <p class="actions">
          <button type="submit" class="button primary">Send the request</button>
          <button type="reset" class="button">Clear the form</button>
        </p>
      </fieldset>
    </form>
```

***Listing 2.7**: UU PDP No. 27/2022 asks for specific, informed, freely given consent — a box the customer ticks after reading what is stored and why, never a pre-ticked one. Book: Section 2.12.8.*

Worked Example: What Siti Sends

> **WORKED EXAMPLE.** Siti fills the pickup form: her name, number and address, Keputih, *Wash, dry & fold*, 4.3 kg, *No perfume* and *Shirts on hangers*, 2 October 2026 at 09:30, a note, consent ticked.

The browser sends a **flat list of name-and-value pairs**, not a form.

`care` appears **twice**, once per ticked box.

`email` appears as an empty string, because an empty text box still sends its name.

`consent=yes` — and, left unticked, the pair would simply not be there.

The radio group is required through `required` on its first button, and `step="0.1"` accepts 4.3 but not 4.35. Book: Sections 2.12.8 and 2.13.3.

A Real GET, on the Wire

Listing 2.8 From `captures/request-1.txt`: the request line and some headers of a real GET sent by the status form.

```
GET /status?bill=B2938&phone=0896+0000+1482 HTTP/1.1
Host: localhost:8155
Connection: keep-alive
Referer: http://localhost:8155/status.html
Accept-Encoding: gzip, deflate, br, zstd
```

`GET` is the method; `/status` is the form's `action`; everything after `?` is the **query string** — the form's data.

Pairs are joined by `&`, name and value by `=`, and the spaces of the phone number became `+`. The body of a GET is empty.

***Listing 2.8**: the capture was made by `tools/capture.py`, a tiny server on a raw socket that prints every request exactly as it arrived; the omitted headers are browser hints. Book: Section 2.13.1.*

Percent-Encoding, Worked by Hand

DEFINITION. Percent-encoding replaces every byte that is not a letter, a digit or one of `-`, `.`, `_`, `~` by `%` and two hexadecimal digits. In form data a space may be `+`.

$$v = 16h + \ell, \qquad h = \lfloor v/16 \rfloor, \quad \ell = v - 16h \tag{2.2}$$

WORKED EXAMPLE. `#` is byte $35 = 2 \times 16 + 3$, so `%23`. The comma is $44 = 2 \times 16 + 12$, and 12 is `C`: `%2C`. The colon is $58 = 3 \times 16 + 10$: `%3A`, so `09:30` travels as `09%3A30`.

By Equation (2.2) a literal `+` must itself be sent as `%2B`, or a search for `C++` becomes `C` and two spaces; `é` is two UTF-8 bytes, `%C3%A9`. Book: Section 2.13.2.

A Real POST, on the Wire

Listing 2.9 From `captures/request-2.txt`: the pickup form sent with POST. The data moved into the body; its format did not change.

```
POST /pickup HTTP/1.1
Content-Length: 258
Origin: http://localhost:8155
Content-Type: application/x-www-form-urlencoded

name=Siti+Rahmawati&phone=0812+0000+1234&email=&address=Jl.+Keputih+Tegal+Timur+12%2C+Sukolilo&outlet=O02&service=
S0001&amount=4.3&care=no-perfume&care=hang-shirts&pickup_date=2026-10-
02&slot=09%3A30&notes=Green+gate%2C+ring+twice+%233&consent=yes&source=web
```

***Listing 2.9**: `Content-Type` says how to read the body and `Content-Length: 258` how many bytes to read; `Origin` names the form's site. The comma arrived as `%2C`, `#3` as `%233`, and `care` twice. Book: Section 2.13.3.*

The Server Receives Pairs

> **KEY INSIGHT.** A name may appear zero times, once, or many times, so every toolkit offers two readers: the first value, and all of them.

Java Servlet: `getParameter` and `getParameterValues`.

Flask: `request.form.get` and `request.form.getlist`.

PHP: `$_POST` keeps only the *last* value unless the HTML names the field `care[]`.

Files: `enctype="multipart/form-data"` sends parts separated by a random **boundary**, each with its own headers and raw bytes.

Using the singular reader on a checkbox group is a bug that only appears when a visitor ticks two boxes. Book: Sections 2.13.3 and 2.13.4.

GET or POST?

Table 2.5 GET against POST. The last row is the one that decides.

	GET	POST
Where the data goes	in the URL, after `?`	in the request body
Visible in the address bar	yes	no
Can be bookmarked or shared	yes	no
Kept in browser history	yes	no
Written to server logs	yes, in full	no, only the path
Size limit	a few thousand characters in practice	effectively none
Can carry files	no	yes
Re-sent by pressing Reload	silently	only after a warning
Cached by browsers and proxies	yes	no
Meant for	asking for something	changing something

***Table 2.5**: GET to read, POST to change. The test that never fails: would it be harmful to do this twice? And never put a password in a GET. Book: Section 2.13.5.*

Validation: Three Places, Three Jobs

In the HTML — `required`, `pattern`, `min`, `type="email"`: free, works without JavaScript, but runs on the visitor's machine.

In JavaScript — every bad field marked at once, in our words, read aloud by a screen reader; switched off in a second.

On the server — the only check that is real.

> **THE MOST IMPORTANT WARNING OF THE CHAPTER.** Client-side validation is a convenience for honest visitors. Anyone can send a request that never passed through the form, with `curl` or twenty lines of Python. The server repeats every check, on every request.

In one line: the client's checks stop mistakes; the server's checks stop attacks. Book: Section 2.14.

CSS: Rules, and Where They Live

```
hr { color: red; height: 5px; width: 50%; }
```

Selector `hr`; **declarations** `property: value;` separated by semicolons. Write the last semicolon too.

Inline — `style="…"` on one tag: almost never. **Internal** — a `<style>` block in the head: one page.

External — `<link rel="stylesheet" href="css/style.css">`: one file for every page, downloaded once and cached. This is the one to use.

CSS has one comment form, `/* … */`; `//` does not work. `h1, h2, h3, h4 { color: blue; }` gives four selectors one declaration. Book: Sections 2.15.1 and 2.15.2.

The Selectors You Will Use

Table 2.6 The CSS selectors used most. A space between two selectors means "inside"; a `>` means "directly inside".

Selector	Matches
`p`	every `<p>` element
`.field-error`	every element whose `class` contains `field-error`
`#price-table`	the one element whose `id` is `price-table`
`h1, h2, h3`	every h1, **or** h2, **or** h3
`nav a`	every `<a>` anywhere inside a `<nav>`
`nav > ul`	every `<ul>` that is a **direct child** of a `<nav>`
`input[type="radio"]`	every input whose `type` attribute is exactly `radio`
`a:hover`	a link while the pointer is over it
`:focus-visible`	the element that has the keyboard focus, when a ring helps
`tr:nth-child(even)`	every second row

***Table 2.6**: a space means "inside", `>` means "directly inside"; `table.receipt` is a table of that class, `table .receipt` anything of that class inside a table. Book: Section 2.15.3.*

`id`, `class`, and a Silent Bug

stylecss04.html

```
<!-- stylecss04.html -->
<HTML>
  <HEAD>
    <TITLE>Profitina Laundry</TITLE>
    <STYLE TYPE="TEXT/CSS">
      #TOP {
        BACKGROUND-COLOR: #FF9;
        PADDING: 1em;
        TEXT-ALIGN: CENTER;
      }
      .MyLine {
        COLOR: BLUE;
        FRONT-WEIGHT: BOLD;
        MARGIN: 0;
        PADDING: 0;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <DIV ID="TOP">
      <H1>Profitina Laundry</H1>
      <P CLASS="MyLine">Clean laundry, three outlets in Surabaya</P>
      <P CLASS="MyLine">Open every day, 07:00 to 21:00</P>
    </DIV>
    <P>Welcome to our website!</P>
  </BODY>
</HTML>
```

Figure 2.3 A page using both kinds of selector (`html/basics/historic/stylecss04.html`): `#TOP` styles one `<div>`, and `.MyLine` styles the two paragraphs that carry that class.

***Figure 2.3**: `#` selects the one unique `id`, `.` any number of elements of a class. `FRONT-WEIGHT: BOLD` is no property, so it is dropped without a word. Prefer classes. Book: Section 2.15.4.*

Color and Length

`#RRGGBB` is three bytes, `00` to `FF` each; `#FF9` doubles each digit to `#FFFF99`; `rgb(37 99 235 / 0.35)` adds transparency.

> **WORKED EXAMPLE.** The button blue `#2563eb`: red `25` $= 2 \times 16 + 5 = 37$, green `63` $= 6 \times 16 + 3 = 99$, blue `eb` $= 14 \times 16 + 11 = 235$. Little red, some green, nearly full blue.

`rem` is the root font size and never compounds; `em` is this element's, so three nested lists at `0.9em` give $0.9^3 \approx 0.73$.

Prefer `rem` for type and spacing, `px` for borders, `%` and `fr` for layout.

The 216 web-safe colors, digits drawn from 0, 3, 6, 9, C and F, belong to old screens and old stylesheets. Book: Sections 2.15.5 and 2.15.6.

Custom Properties: Name a Design Once

> **DEFINITION.** A **custom property** begins with two hyphens, is usually declared on `:root`, and is read with `var(--blue)`; `var(--space, 1rem)` gives a fallback.

Listing 2.10 From `html/css/style.css`: a dark theme that changes only the design tokens.

```
@media (prefers-color-scheme: dark) {
  :root {                              /* only the tokens change; no rule below is touched */
    --ink: #e5e7eb;
    --muted: #9ca3af;
    --line: #374151;
    --bg: #111827;
    --card: #1f2937;
    --blue-dark: #93c5fd;
    --bad: #fca5a5;
    --bad-bg: #3b1d1d;
  }
}
```

***Listing 2.10**: the rest of the stylesheet never names a color — `body { color: var(--ink); background: var(--bg); }` — so eight new token values re-theme the whole site. Book: Section 2.16.*

Specificity Is Arithmetic

> **DEFINITION.** The **specificity** of a selector is (a, b, c): a ids; b classes, attribute selectors and pseudo-classes; c element names and pseudo-elements. Compare left to right.

For every selector a human would write, Equation (2.3) gives the right order.

$$\text{score} = 100a + 10b + c \tag{2.3}$$

Stage 1 origin and `!important`; **stage 2** specificity; **stage 3** the later rule wins a tie; **stage 4** inheritance (text properties inherit, box properties do not).

It is not really a sum: one id beats any number of classes, because the comparison is by column. `!important` is a last resort. Book: Sections 2.17.1 and 2.17.2.

Specificity for Real Selectors

Table 2.7 Specificity computed for real selectors from the stylesheet. Count the ids, then the classes, attributes and pseudo-classes, then the element names.

Selector	Ids (*a*)	Classes (*b*)	Elements (*c*)	Triple	Score
`*`	0	0	0	(0,0,0)	0
`a`	0	0	1	(0,0,1)	1
`.card`	0	1	0	(0,1,0)	10
`.site-menu a`	0	1	1	(0,1,1)	11
`#price-table`	1	0	0	(1,0,0)	100
`#price-table tbody th`	1	0	2	(1,0,2)	102
a `style` attribute	—	—	—	—	beats all

***Table 2.7**: one id at 100 beats nine classes at 90; an inline `style` beats every selector; `*` scores zero, which is what a reset wants. Book: Section 2.17.2.*

Worked Example: Which Color Wins?

Listing 2.11 Four rules competing for the color of one link (`html/basics/cascade.css`).

```
a                                { color: #0000ee; }   /* (0,0,1) =  1 */
.site-nav a                      { color: #0f766e; }   /* (0,1,1) = 11 */
nav ul li a                      { color: #ff0000; }   /* (0,0,4) =  4 */
.site-nav a[aria-current="page"] { color: #1d4ed8; }   /* (0,2,1) = 21 */
```

The link is `<a href="#" aria-current="page">Order</a>` inside `<nav class="site-nav">`. Scores: 1, 11, 4, 21.

The fourth rule wins with 21: `#1d4ed8`. Chromium answers `rgb(29, 78, 216)` — $29 = 1 \times 16 + 13$, $78 = 4 \times 16 + 14$, $216 = 13 \times 16 + 8$.

> **KEY INSIGHT.** A rule that "does not work" has usually worked and been beaten. The Styles panel strikes the loser through: it is the cascade, drawn.

***Listing 2.11**: `nav ul li a`, four elements, loses to `.site-nav a`, because b is compared before c. Book: Section 2.17.3.*

The Box Model

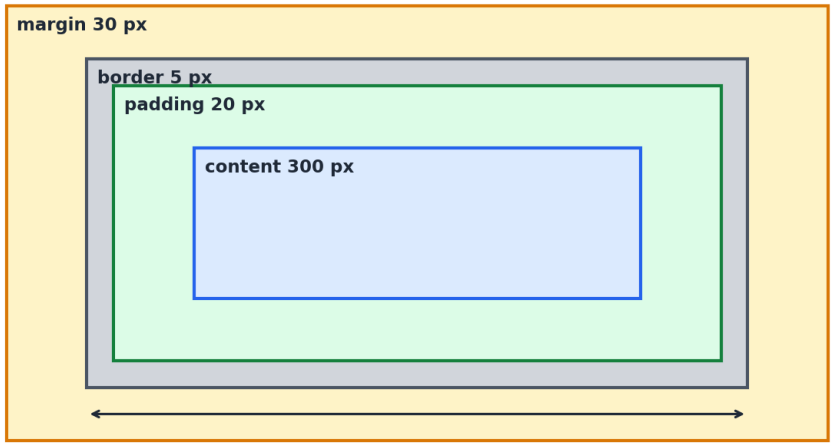


Figure 2.4 The box model of a card with `width: 300px`, 20 pixels of padding, a 5-pixel border and a 30-pixel margin. Content in the middle, then padding, border and margin.

***Figure 2.4**: content, then padding (inside, with the background), border, and margin (outside, transparent). Book: Section 2.18.*

Worked Example: 300 Pixels Is 410

Listing 2.12 From `html/basics/box-model.css`: one card, with width, padding, border and margin.

```
.card {
  width: 300px;
  padding: 20px;
  border: 5px solid #d1d5db;
  margin: 30px;
}
```

$$\text{visible width} = w + 2p + 2b = 300 + 2 \times 20 + 2 \times 5 = 350 \text{ px} \tag{2.4}$$

With margins: $350 + 2 \times 30 = 410$ px occupied. Under `box-sizing: border-box` the card is exactly 300 px and its content shrinks to $300 - 40 - 10 = 250$.

Listing 2.12 *and Equation (2.4): Chromium measures the two cards at 350 and 300 pixels. Write* `*, *::before, *::after { box-sizing: border-box; }` *once; vertical margins between blocks collapse to the larger. Book: Section 2.18.*

Flexbox: One Dimension

Listing 2.13 From `html/css/style.css`: the page header and the menu laid out with flexbox.

```
/* ---- 3. Header: Flexbox (one dimension, a row that may wrap) ---- */
.site-header {
  display: flex;
  flex-wrap: wrap;
  align-items: center;
  justify-content: space-between;
  gap: 0.5rem 1.5rem;
  padding: 0.6rem var(--space);
  background: var(--card);
  border-bottom: 3px solid var(--teal);
}
.site-menu ul { display: flex; flex-wrap: wrap; gap: 0.25rem; list-style: none;
                margin: 0; padding: 0; }
```

***Listing 2.13**: `justify-content` works along the line, `align-items` across it; `flex-wrap: wrap` lets the menu drop under the logo on a phone; three declarations turn a `<ul>` into a horizontal menu. Book: Section 2.19.1.*

Grid, and Mobile First

Grid lays out two dimensions: `grid-template-columns: repeat(auto-fit, minmax(14rem, 1fr));` makes as many columns as fit, each at least 14 rem.

Listing 2.14 A media query (excerpt of `html/css/style.css`). Below 48 rem the calculator sits under the price table; from 48 rem it sits beside it.

```
/* ---- 8. Wider screens: mobile first, so the query ADDS layout ---- */
@media (min-width: 48rem) {
  h1 { font-size: 2rem; }
  .hero { padding: 2.5rem; }
  .columns {                          /* price table and calculator side by side */
    display: grid;
    grid-template-columns: 3fr 2fr;
    gap: 1.5rem;
    align-items: start;
  }
}
```

***Listing 2.14** is **mobile first**: the narrow layout needs no query, and the query adds. Book: Sections 2.19.2 and 2.20.*

JavaScript: Behavior

Table 2.8 JavaScript compared with Java. Beyond four letters of the name, they have little in common.

	JavaScript	Java
Runs	in the browser, and on servers through Node.js	on servers, desktops and phones
Executed by	the browser's engine, straight from source	the Java Virtual Machine, from compiled bytecode
Created by	Netscape	Sun Microsystems
Its source	visible to anyone who opens the page	compiled; the source is not shipped
Its relation to HTML	written inside or beside the document	a separate program that may produce the document

It responds to events, reads and changes the page through the DOM, validates, talks to a server with `fetch`, and remembers things in the browser.

***Table 2.8**: written in ten days in 1995 and standardized as ECMAScript. Its source is visible to anyone, so it can never be trusted — Section 2.14 from the other side. Book: Sections 2.21 and 2.21.1.*

The Language in One Sitting

Listing 2.15 From `html/basics/js-basics.js`: `const`, `let`, a template literal and the four classic surprises. Run it with `node js-basics.js`.

```
const shop = "Profitina Laundry";          // const: this name never points elsewhere
let kg = 4.3;                              // let: this one may change
console.log(typeof shop, typeof kg, typeof true, typeof undefined);

kg = Math.ceil(kg);                        // 4.3 -> 5: we round up at the counter
const pricePerKg = 8000;
console.log(`${shop}: ${kg} kg x ${pricePerKg} = ${kg * pricePerKg}`);

console.log("5" + 3, "5" * 3, "5" === 5, 0.1 + 0.2);   // the four classic surprises
```

```
string number boolean undefined
Profitina Laundry: 5 kg x 8000 = 40000
53 15 false 0.30000000000000004
```

***Listing 2.15**: always use `===`; `0.1 + 0.2` is not `0.3`, so money is counted in whole rupiah. Book: Section 2.21.2.*

The DOM, Walked

Listing 2.16 From `html/basics/dom-walk.html`: a recursive walk that writes one line per element.

```
    function walk(node, depth, lines) {
      lines.push("  ".repeat(depth) + node.tagName.toLowerCase());
      for (const child of node.children) walk(child, depth + 1, lines);
      return lines;
    }
    const lines = walk(document.documentElement, 0, []);
    document.getElementById("tree").textContent = lines.join("\n");
```

```
html
  head
    meta
    title
  body
    h1
    p
      a
    pre
    script
```

***Listing 2.16**: the document tree turned sideways, `<script>` included. Book: Section 2.22.*

Changing the Page Safely

Listing 2.17 Rebuilding the rows of the price table with the DOM (excerpt of `html/js/prices.js`).

```
/** Replaces the rows of the price table with one row per service. */
function showTable(services) {
  const tbody = document.querySelector("#price-table tbody");
  const rows = services.map(function (s) {
    const tr = document.createElement("tr");
    const th = document.createElement("th");
    th.scope = "row";
    th.textContent = s.description;              // textContent: shown as text, never as markup
    const td = document.createElement("td");
    td.className = "num";
    td.textContent = Laundry.money(s.price);
    tr.append(th, td);
    return tr;
  });
  tbody.replaceChildren(...rows);
}
```

> **NEVER `innerHTML` WITH DATA.** A description such as `<img src=x onerror=…>` would become live code: cross-site scripting. Create elements and set `textContent`.

Listing 2.17 *uses the five operations: find (`querySelector`), text (`textContent`), attributes (`className`, `scope`), create (`createElement`), insert (`append`, `replaceChildren`). Book: Section 2.22.1.*

Events

```
form.addEventListener("input", update);
```

Listing 2.18 The calculator's listener (excerpt of `html/js/prices.js`). It reads the chosen service and the amount, and writes one line of text.

```
  function update() {
    const price = Number(select.selectedOptions[0].dataset.price);
    const units = Laundry.roundUpUnits(amount.value);
    if (units === null || units > 50) {
      result.textContent = "Enter a weight or a number from 0.1 to 50.";
      return;
    }
    result.textContent = Laundry.priceLine(price, units);
  }
```

***Listing 2.18**: events **bubble**, so one listener on the whole form hears both controls; 4.3 kg becomes `5 x Rp 8.000 = Rp 40.000`. `event.preventDefault()` cancels what the browser would do next. Book: Section 2.22.2.*

A Form, or `fetch`?



Figure 2.5 Two ways to ask the server about bill B2938. A form loads a whole new page; `fetch` gets JSON and the script changes one part of the current page.

***Figure 2.5**: a form loads a whole new page; `fetch` gets JSON, and the script changes one part of the current page. Book: Section 2.23.*

`fetch`, JSON, and a Fallback

Listing 2.19 Fetching the price list as JSON, with a fallback (excerpt of `html/js/prices.js`).

```
/** The price list: from the server's API if one answers, else from a static file. */
async function loadServices() {
  for (const url of ["api/services", "data/services.json"]) {
    try {
      const response = await fetch(url, { headers: { Accept: "application/json" } });
      if (!response.ok) continue;                 // 404: no API here, try the next address
      const body = await response.json();         // text -> JavaScript objects
      return { services: body.data, url: url };
    } catch (err) {
      // no network, or not JSON: try the next address
    }
  }
  return null;                                    // keep the table written in the HTML
}
```

***Listing 2.19**: `await` waits without freezing the page; `response.ok` is true for 200 to 299; if both addresses fail, the table written in the HTML stays. The page works without the script — **progressive enhancement**. Book: Section 2.23.*

An Error Status Is an Answer

Listing 2.20 Asking `GET /api/status` for JSON and showing the answer (excerpt of `html/js/status.js`).

```
async function check() {
  result.hidden = true;
  const query = new URLSearchParams(new FormData(form));   // bill=B2938&phone=0896...
  try {
    const response = await fetch("api/status?" + query, { headers: { Accept: "application/json" } });
    const body = await response.json();
    if (response.ok) {
      say("");
      show(body);
    } else {
      say(body.error.message);                    // 400 or 404: the server says why
    }
  } catch (err) {
    say("The status check needs one of the chapter's servers (java, python or php).");
  }
}
```

***Listing 2.20**: `fetch` throws only when there is no answer at all; a 404 carries `{"error": {"code": "not_found", …}}` and the page shows the server's message. The answer holds no name or address: data minimization, decided on the server. Book: Section 2.23.*

Structure, Presentation, Behavior

Version 1 (1999): `<font>`, `<center>`, a layout table, `Arial` typed four times, and hover done by two JavaScript attributes.

Version 2: semantic elements, all appearance in one `<style>` block, hover by `a:hover`.

Version 3: structure in the HTML, presentation in `home.css`, behavior in `home.js`.

Listing 2.21 From `html/steps/home.js`: the rate is read from `data-price` in the page, so the number 8000 exists once.

```
// The price lives in the HTML, on the list item, not here.
const item = document.querySelector("#price-list li[data-price]");
const rate = item ? Number(item.dataset.price) : 0;

function recompute() {
  const units = Math.ceil(Number(kg.value));  // 2.5 kg is charged as 3 kg
  out.textContent = (rate * units).toLocaleString("id-ID");
}
kg.addEventListener("input", recompute);
recompute();
```

***Listing 2.21**: delete `home.js` and the page still reads perfectly; delete `home.css` too and it is still a document with a heading, a list and a link. Each layer degrades to something useful. Book: Section 2.24.*

The Public Pages of Profitina Laundry

Table 2.9 The public pages of this chapter and their counterparts in the finished front end.

Chapter 2 file	What it does	In App/static/
index.html, js/prices.js	price table, calculator	index.html, js/home.js
status.html, js/status.js	“Where is my laundry?”	status.html, js/status.js
pickup.html, js/pickup.js	pickup request form	(taken by phone in the shop today)
js/laundry.js	the shop’s rules, also run by Node.js	js/core.js
css/style.css	tokens, flexbox, grid, media query	css/style.css
data/services.json	price list without a server	../database/csv/ (mock mode)

***Table 2.9**: seven steps — skeleton and header, content in HTML alone, the stylesheet, behavior, the status page, the pickup form, and tests run by Node.js (`node --test`), nine of nine passing. Book: Section 2.25.*

The Finished Price Page

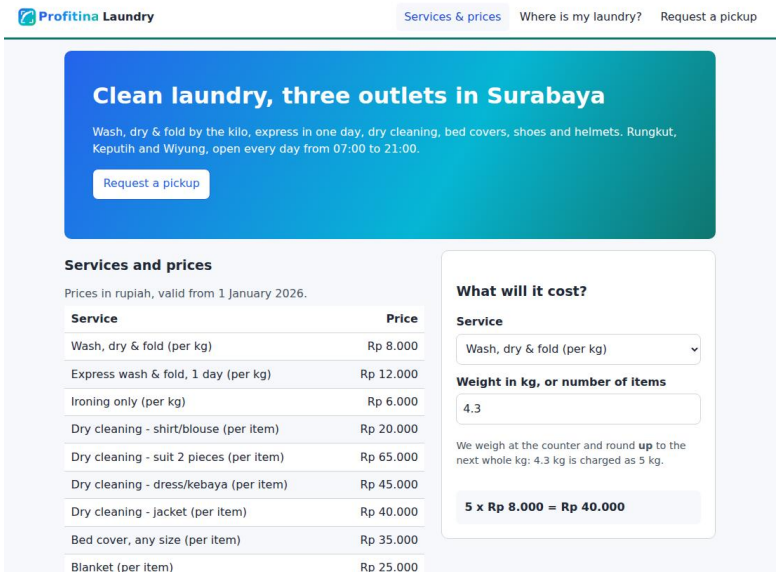


Figure 2.6 The finished price page. The table came from `GET /api/services`, and the calculator has already priced 4.3 kilograms as five.

***Figure 2.6**: the table came from `GET /api/services`; with the stylesheet and scripts removed the page is still a complete price list. Book: Section 2.25.*

Stopping a Bad Pickup Request

Listing 2.22 Checking the pickup form before it is sent (excerpt of `html/js/pickup.js`).

```
  form.addEventListener("submit", function (event) {
    const errors = Laundry.validatePickup(values(), serviceIds);
    showErrors(errors);
    const first = Object.keys(errors)[0];
    if (first) {
      event.preventDefault();                    // stop the request leaving the browser
      const control = form.elements[first];
      (control.length && !control.options ? control[0] : control).focus();
    }
  });
```

***Listing 2.22**: `values()` reads the checkbox group with `getAll("care")`; each message goes into the field's error span with `aria-invalid`; `preventDefault()` stops the request and the first bad field gets the focus. Book: Section 2.25.*

Three Servers, One Parser

Java `java LaundryServer` (8080), Flask `python app.py` (5000), PHP `php -S localhost:8000 -t ../html router.php` (8000): the same pages, the same answers.

Listing 2.23 The form parser (excerpt of `java/LaundryServer.java`). Split on `&`, split each pair on the first `=`, percent-decode both halves, and append, because a name may repeat.

```java
    /** "a=1&b=x+y&b=%26" -> {a=[1], b=[x y, &]}: a name may appear many times. */
    static Map<String, List<String>> parse(String raw) {
        Map<String, List<String>> out = new LinkedHashMap<>();
        if (raw == null || raw.isEmpty()) return out;
        for (String pair : raw.split("&")) {
            int eq = pair.indexOf('=');
            String k = eq < 0 ? pair : pair.substring(0, eq);
            String v = eq < 0 ? "" : pair.substring(eq + 1);
            k = URLDecoder.decode(k, StandardCharsets.UTF_8);
            v = URLDecoder.decode(v, StandardCharsets.UTF_8);
            out.computeIfAbsent(k, x -> new ArrayList<>()).add(v);
        }
        return out;
    }
```

Listing 2.23 *is Section 2.13 turned into code: a value may contain* `=`*, and each name keeps a* ***list****. PHP does the same with* `explode`*,* `urldecode` *and* `$out[$k][] = $v`*. Book: Sections 2.26.1 and 2.26.3.*

Flask Does the Parsing

Listing 2.24 From `python/app.py`: the JSON status check, with `400` and `404` answers.

```
@app.get("/api/status")
def api_status():
    bill, phone = request.args.get("bill", ""), request.args.get("phone", "")
    if not bill.strip() or not digits(phone):
        return jsonify(error={"code": "bad_request",
                              "message": "Give the bill number and the phone number."}), 400
    b = find_bill(bill, phone)
    if b is None:
        return jsonify(error={"code": "not_found",
                              "message": "No bill found for this number and phone."}), 404
    return jsonify(status_of(b))
```

Listing 2.25 Reading a checkbox group: `getlist`, not `get`, and then discarding anything that is not on the list (excerpt of `python/app.py`).

```
    care = [CARE[c] for c in f.getlist("care") if c in CARE]    # getlist: every ticked box
```

***Listing 2.24**: `request.args` is the parsed query. **Listing 2.25**: `getlist`, and then only known values — `care=free` from `curl` is dropped. A bad pickup gets `422`. One smoke test sends twelve requests; all three servers pass all twelve. Book: Sections 2.26.2 and 2.26.4.*

Profitina in Practice

Structure is the signal: from `<div>` soup there is little to extract; from `<main>`, headings and paragraphs an answer engine can quote.

The page must work before the script runs: prices that exist only after JavaScript may be invisible to a crawler.

Extractability overlaps with accessibility: `alt`, one `<h1>`, headings in order, `<th scope>` — what a screen reader can parse, a language model can parse.

Structured data is markup too: `schema.org` and `robots.txt`, drafted by a model running on Profitina's own server.

The runnable case is always Profitina Laundry. Book: Section 2.27.

Common Mistakes

A control with no `name`; GET that changes data; the singular reader on a checkbox group; expecting an unticked box to send `""`.

Trusting the browser's validation; a visitor's text in a page without escaping; `innerHTML` with data.

A heading chosen for its size; a table for layout; `<div>` where a real element exists; one `id` twice.

No `border-box`; `!important` instead of reading the specificity; a color typed twelve times.

`==` in JavaScript; forgetting that `fetch` does not fail on 404; never running the validator.

Sixteen mistakes are listed in the book, each with its fix. Book: Section 2.28.

Summary: Five Things to Remember

1. HTML describes what things *are*; a page is a tree that starts with `<!DOCTYPE html>`, `lang`, `charset`, viewport and a real `<title>`.

2. A form sends name-and-value pairs: no `name`, no value; GET reads, POST changes; both percent-encode (Equation (2.2)).

3. Validate in the HTML, in JavaScript, and on the server — the only check that can be trusted.

4. CSS conflicts are settled by importance, specificity (Equation (2.3)), then order; every element is a box (Equation (2.4)); flexbox, grid and media queries lay out, mobile first.

5. JavaScript changes the page through the DOM, with `textContent`, and talks to servers with `fetch` — the page works without it.

Structure in the HTML, presentation in the CSS, behavior in the JavaScript. Book: Section 2.29.

Next Time

Chapter 3 — PHP from Zero. The file with pockets of code, variables and types, operators, decisions and loops, functions, arrays and strings.

Superglobals, forms, validation and escaping, cookies and sessions, passwords, file uploads — with the same pages in Java and Flask.

Before then: work Exercises 2.3, 2.8, 2.9, 2.11 and 2.18, and run the W3C checker on a page of your own.

Everything from Chapter 3 on is code that reads the pairs we sent today. Book: Sections 2.30 and 2.31.

Lecture 3 — PHP from Zero

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 3

Lecturer: Irfan Subakti — yifana@gmail.com

Today the server starts to think: we write PHP 8.4 from the first line, read what the visitor sent, refuse what is bad, and remember the visitor between requests — with every web idea shown in PHP, Java and Flask. Book: Chapter 3, Sections 3.1–3.27.

What This Meeting Covers

1. **The server-side idea** — a program in the middle of the request; Hello World from three servers.
2. **The language** — the shape of a file, variables and types, juggling and `===`, operators and precedence, decisions, loops, functions, arrays, strings, `require_once`.
3. **The request** — superglobals, GET and POST, forms, and the price calculator in PHP, Java and Flask.
4. **The discipline** — validation and escaping, cookies and sessions, the form token, post/redirect/get, password hashes, file uploads, reading errors.
5. **Profitina Laundry** — a calculator, a contact form and a staff page that reads the messages.

Section 3.19, validation and escaping, is the most important part of today. Book: Section 3.1.

Where We Left Off: the Same for Everybody

Chapter 1: **a browser asks, a server answers**. Chapter 2: HTML and CSS worth drawing.

But every page was **the same for everybody**: a new price meant editing the file, and a submitted form reached nothing that listened.

A static page can say “we charge Rp 8.000 per kilogram”. It cannot say “Sri, four kilograms will cost Rp 32.000”.

Three missing things: **a place for logic**, **a way to receive what the visitor typed**, and **the discipline that must come with it**.

The moment a stranger can send data into a program, its assumptions become a target. Book: Section 3.1.

A Program in the Middle

> **DEFINITION.** A **static page** is a page whose bytes are decided before anybody asks for it: the file on disk and the body of the response are the same thing.

For `.php` the server hands the file to the **PHP interpreter**, waits, and sends back whatever it printed.

> **KEY INSIGHT.** PHP does not run in the browser. It runs on the server, finishes, and is gone; "View source" shows only the HTML it printed.

PHP, JSP, ASP.NET and Python sit in the same slot; PHP has the shortest road from nothing to a working page.

Java and Python come with PHP in every chapter, so that we learn the idea rather than the dialect. Book: Sections 3.2.1–3.2.3.

The PHP Architecture

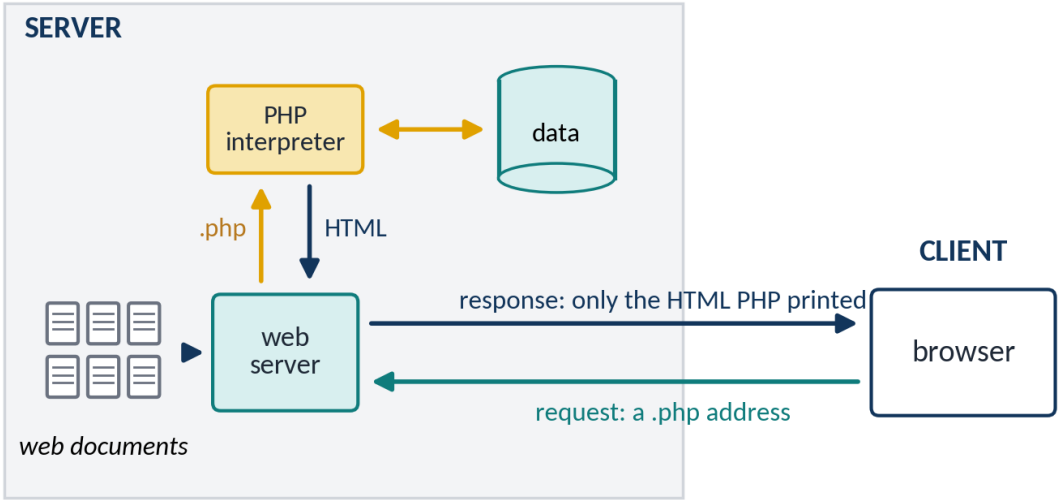


Figure 3.1 The PHP web application architecture. The server passes the file to the PHP interpreter, which may consult data and produces the HTML response. The client never sees the PHP, only what it printed.

Figure 3.1*: the client sends an ordinary request; the server starts the interpreter; the program runs top to bottom, perhaps reading data; its output becomes the response. Book: Section 3.2.2.*

The Life of One Request

Table 3.1 The life of one request. Steps 5 to 8 are this chapter; the rest was Chapter 1.

Step	Who does it	What happens
1	Browser	You click a link or press a submit button
2	Browser	It builds an HTTP request: a method, a path, headers, sometimes a body
3	Network	The request travels to the server named in the address
4	Web server	It receives the request and looks at the path
5	Web server	The extension is `.php`, so it starts the PHP interpreter on that file
6	PHP interpreter	It fills the superglobals with what the browser sent (Section 3.17)
7	PHP interpreter	It runs your program, top to bottom, once
8	PHP interpreter	Everything printed is gathered into the response body
9	Web server	It adds the status line and headers and sends the response
10	Browser	It parses the HTML, fetches the CSS and images, and draws the page

***Table 3.1**: the interpreter runs the program **once, then forgets everything**, so what must survive goes into a file, a database or a session; and the program never talks to the browser — it prints, the server delivers. Book: Section 3.2.4.*

Hello World, Through a Server

Listing 3.1 `hello.php`: HTML on the outside, a single PHP instruction on the inside.

```
<html>
    <body>
        <?php
            echo "Hello World!";
        ?>
    </body>
</html>
```

```
<html>
    <body>
        Hello World!    </body>
</html>
```

Everything outside `<?php … ?>` is copied untouched: a PHP file is an HTML file with holes cut in it. Every statement ends with `;`.

***Listing 3.1**: run `php -S 127.0.0.1:8080` in the folder and request it over `http://`; a `file:///` address never runs the PHP. A file that is pure PHP has no closing `?>`. Book: Sections 3.2.5 and 3.3.3.*

Hello World from Three Servers

Listing 3.2 `hello.py`: a Flask application. The decorator `@app.route("/")` binds the address to the function below it.

```
"""hello.py — the smallest Flask application."""
from flask import Flask

app = Flask(__name__)


@app.route("/")                        # a URL is bound to a FUNCTION
def hello():
    return "<html>\n\t<body>\n\t\tHello World!\n\t</body>\n</html>\n"


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

In PHP **a URL maps to a file**; in Java to **a handler** registered with `createContext`; in Flask to **a function** through `@app.route`.

***Listing 3.2**: PHP supplied `200 OK` and `Content-Type` by itself; the JDK server made us write the status, the headers and the body length; Flask sat in between, filling in the rest of a returned string. Book: Section 3.3.4.*

The Ways to Open a Block

Opening … closing	Name	In PHP 8.4
`<?php … ?>`	standard tag	always works: use it
`<?= … ?>`	short echo	always works: same as `<?php echo …; ?>`
`<? … ?>`	short open tag	only if short_open_tag=On: avoid
`<script language="php"> … </script>`	script tag	removed in PHP 7
`<% … %>`	ASP-style tag	removed in PHP 7

Figure 3.2 The opening and closing tags PHP has accepted. Only `<?php … ?>` and its echo shorthand `<?= … ?>` work on every server; the others are optional or removed.

***Figure 3.2**: use `<?php … ?>` and its echo shorthand `<?= $x ?>`; the short tag depends on a setting, and the script and ASP forms were removed in PHP 7. Book: Section 3.4.1.*

Compute at the Top, Print in the Middle

Listing 3.3 `mixing.php`: compute at the top, print in the middle. The two `<?= … ?>` holes are filled when the page is built.

```php
<?php
$customer = "Sri Wahyuni";
$kilos    = 3;
?>
<!DOCTYPE html>
<html lang="en">
<body>
  <h1>Profitina Laundry</h1>
  <p>Customer: <?= $customer ?></p>
  <p>Weight: <?= $kilos ?> kg</p>
</body>
</html>
```

Listing 3.3 *sends* `<p>Customer: Sri Wahyuni</p>` *and* `<p>Weight: 3 kg</p>`*: no PHP in the response, because PHP has already been and gone. Comments:* `//`*,* `#`*,* `/* … */`*. Names we invent are case sensitive; names PHP provides are not. Book: Sections 3.4.2–3.4.4 and 3.5.*

Variables and Constants

A variable starts with `$`, needs no declaration, is case sensitive, and is created by assignment — so `$totl = $total + 2000;` silently makes a new one.

> **DEFINITION.** An **assignment** `$name = expression;` evaluates the right side, then stores the result under the name. `=` means "becomes"; equality is `==` or `===`.

Constants have no `$`: `const PRICE_PER_KG = 8000;` or `define('SHOP_NAME', 'Profitina Laundry');` — global, and edited in one place.

Java declares, `int units = 4;`, and a typo is a compile error; Python, like PHP, creates on assignment but without the `$`. Book: Sections 3.6 and 3.8.

PHP's Types

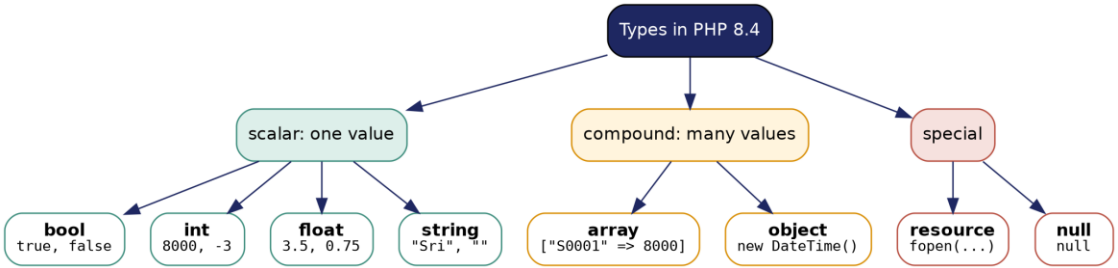


Figure 3.3 PHP's types in three families: scalar types hold one value, compound types hold several, and special types stand for something outside PHP or for no value at all.

***Figure 3.3**: types belong to values, not to names. Floats are binary, so `0.1 + 0.2` is not `0.3` and money never goes in a float; `null` is not `0` and not `""`. Book: Sections 3.7.1 and 3.7.2.*

Type Juggling, and Two Kinds of Equals

`"10" + 5` is `15`: `+` only adds, so the string is read as a number. `"10" . 5` is `"105"`: `.` only joins.

The operator chooses the context, and the context drives the conversion.

`"10" == 10` is `true` (loose); `"10" === 10` is `false` (strict: type first). `"abc" == 0` is `false` in PHP 8, `true` in PHP 7.

Convert on purpose: `intval("3.9 kg")` is 3, `floatval` 3.9, and `is_numeric` honestly says `false`.

> **USE `===` AND `!==`** unless you have a reason not to.

JavaScript uses `+` for both jobs; PHP's separate `.` is one of its better ideas. Book: Sections 3.7.3 and 3.7.4.

What Counts as False

Table 3.2 The complete list of falsy values in PHP. Note `"0"`, which is false, against `"0.0"` and `" "`, which are both true.

Value	As a Boolean
`false`	false
`0` (integer zero)	false
`0.0` and `-0.0`	false
`""` (the empty string)	false
`"0"` (the string holding one zero)	false
`[]` (the empty array)	false
`null`	false
anything else	**true**

***Table 3.2**: `"0"` is false, unlike in Python; a single space is true, so a field holding one space is "true". That is why every validator calls `trim()` first. Book: Section 3.7.5.*

Worked Example: Precedence

Work `2 + 3 * 4 ** 2` by hand, highest precedence first:

> **WORKED EXAMPLE.** `**` binds tightest: $4^2 = 16$, leaving `2 + 3 * 16`. Then `*` beats `+`: $3 \times 16 = 48$. Finally $2 + 48 = 50$ — and `03_operators.php` prints 50.

The trap: `and` binds *less* tightly than `=`, so `$x = true and false;` means `($x = true) and false;` and `$x` is `true`. Write `&&` and `||`.

Short-circuit: in `isset($_POST['units']) && $_POST['units'] > 0` the second test never runs when the field is missing. `??` supplies a default: `$_POST['note'] ?? ''`.

`17 / 5` is `3.4` and `intdiv(17, 5)` is 3; `17 % 5` is 2; `<=>` returns negative, zero or positive, exactly what a sort needs. Book: Section 3.9.

Making Decisions

```
if ($units <= 0) {
    $band = "nothing to wash";
} elseif ($units < 5) {
```

`elseif` takes the **first** true condition and skips the rest, so ordering from narrowest to widest keeps each test simple.

> `=` **IS NOT** `==`**.** `if ($age = 17)` assigns 17, is always true, and destroys the variable. `if (17 == $age)` turns the slip into a syntax error.

The ternary chooses a value; the **alternative syntax** `if (…): … endif;` keeps templates readable.

Every page of the Laundry site uses the alternative syntax, because `endif`, `endforeach` and `endwhile` say what is closing. Book: Sections 3.10.1 and 3.10.2.

How `switch` Runs

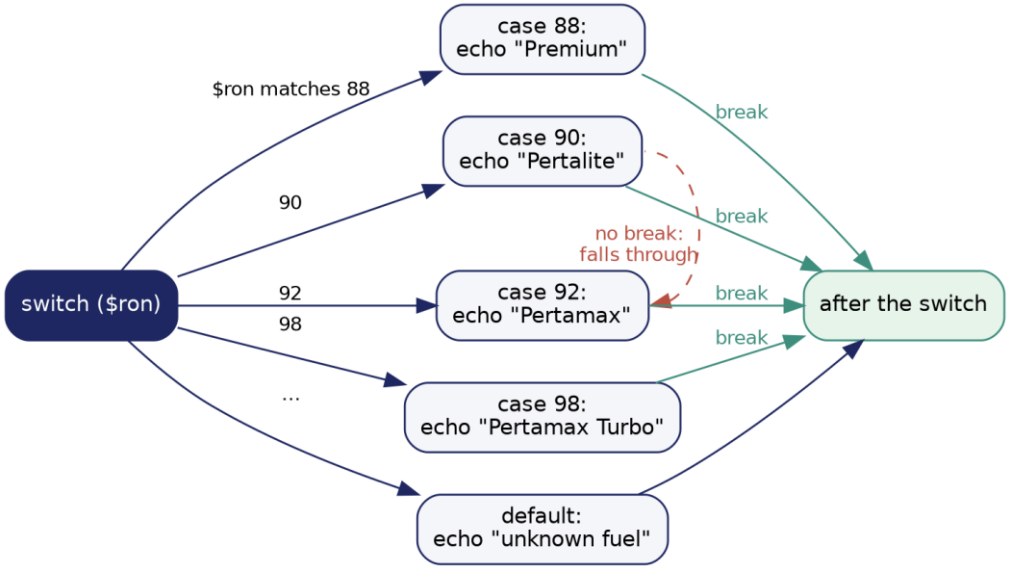


Figure 3.4 How `switch` runs. Control jumps to the matching `case`; `break` leaves the statement. Without `break`, execution *falls through* into the next case.

***Figure 3.4**: with RON 90 matched and no `break`, PHP prints `PertalitePertamaxPertamax Turbo` — it falls through every case below. Book: Section 3.10.3.*

`switch` or `match`?

```
echo match ($ron) {
    88 => "Premium", 90 => "Pertalite", 92 => "Pertamax", 98 => "Pertamax Turbo",
    default => "unknown fuel",
};
```

Table 3.3 The branching forms, and when each is the honest choice.

Form	Use it when	Comparison	Produces
`if` / `if … else`	one or two outcomes	any condition	nothing
`if … elseif … else`	several ordered conditions	any condition	nothing
`?:`	you need to *choose a value*	any condition	a value
`switch`	one value against a list	loose `==`	nothing
`match`	one value against a list	strict `===`	a value

***Table 3.3**: `switch` compares with loose `==` and needs `break`; `match` compares with `===`, needs no `break`, produces a value, and throws when nothing matches without a `default`. Book: Section 3.10.3.*

Four Loops, One Question

The question: **do you know in advance how many times?**

`for` — start, test, step on one line: `for ($i = 1; $i <= 7; $i++)` prints `1234567`.

`while` — when the count is unknown; forget the increment and the page spins until `max_execution_time`.

`do … while` — tests at the bottom, so it runs **at least once**: it prints `100` although `100 <= 7` is false.

`foreach` — walks an array with no index and no way off the end.

Java has the same four; Python has only `for x in …` and `while`, and no `do … while`. Book: Section 3.11.

`break` and `continue`

break_continue.php

```
1  <html>
2      <body>
3          <?php
4              for ($i = 0; $i <= 10; $i++) {
5                  if ($i == 3) {
6                      continue;
7                  }
8                  if ($i == 7) {
9                      break;
10                 }
11                 echo "$i ";
12             }
13         ?>
14     </body>
15 </html>
```

Figure 3.5 `php/classic/break_continue.php`: a `break` and `continue` example. The loop is written to run from 0 to 10, but two statements inside it change that.

Figure 3.5*: the loop is written from 0 to 10, but* `continue` *at 3 and* `break` *at 7 change that. Book: Section 3.11.1.*

Worked Example: the Trace

Table 3.4 The loop of **Figure 3.5** traced by hand. At 3, `continue` jumps past the `echo`; at 7, `break` abandons the loop, so 8, 9 and 10 never happen.

`$i`	`$i == 3`?	`$i == 7`?	What happens	Printed so far
0	no	no	prints `0`	`0`
1	no	no	prints `1`	`0 1`
2	no	no	prints `2`	`0 1 2`
3	**yes**	(not tested)	`continue`: skip to the next pass	`0 1 2`
4	no	no	prints `4`	`0 1 2 4`
5	no	no	prints `5`	`0 1 2 4 5`
6	no	no	prints `6`	`0 1 2 4 5 6`
7	no	**yes**	`break`: leave the loop	`0 1 2 4 5 6`

***Table 3.4**: the output is `0 1 2 4 5 6`. When `continue` fires, the `$i++` of the `for` header still runs; at 7 `break` fires before the `echo`. Book: Section 3.11.1.*

Functions and Scope

`function name(parameters) { …; return value; }` — `return` stops the function and hands the value to the *caller*; it does not print it.

Types in and out: `function circleArea(float $radius): float`; with `declare(strict_types=1);` a wrong type is refused, not converted.

Defaults and named arguments: `laundryCost(units: 4, delivery: 10000)` keeps the default price.

Scope: a variable of the main script is simply not there inside a function. Pass it in; a `static` variable keeps its value between calls, within one request.

A procedure is a function whose `return` you never wrote; it gives back `null`. Book: Sections 3.12.1–3.12.3.

By Value, by Reference, and Recursion

By **value** (the default) the function changes its own copy; with `int &$total` it receives **the variable itself**.

Prefer by value: `addTipByReference($bill);` gives no hint at the call site that `$bill` is about to change.

Arrays are copied on assignment; objects are not.

> **WORKED EXAMPLE.** `factorial(5)` calls `factorial(4)`, down to `factorial(1)`, which returns 1; the answers multiply back up to $1 \times 2 \times 3 \times 4 \times 5 = 120$.

In `05_functions.php` the two `$bill` lines show it: by value the caller's variable survived, by reference it did not. Book: Section 3.12.4.

An Array Is an Ordered Map

> **DEFINITION.** An **array** in PHP is an ordered collection of key–value pairs; each key is an integer or a string, and each value is anything, including another array.

Indexed: `["durian", "mango", "mangosteen", "pineapple"]` counts from **0**; `$a[] = "red";` appends. Asking for `[4]` warns and gives `null`.

Associative: `["Yarik" => 80, "Naya" => 90]`, read *maps to*; insertion order is kept, so `foreach` is predictable.

Keys are unique; `"7"` and `7` are one key; read a missing key with `isset($a['k'])` or `$a['k'] ?? $default`.

With integer keys it is Java's array; with string keys, Python's `dict` or Java's `LinkedHashMap`. Book: Sections 3.13.1–3.13.3.

Walking, Seeing, and Sorting

`foreach ($grade as $name => $grd)` — use `foreach` for arrays and `for` for numbers. `each()` was removed in PHP 8. `print_r()` shows the structure; `var_dump()` adds types and lengths.

Table 3.5 The sorting family. The letter before `sort` tells the story: `a` for associative (keys kept), `k` for key, `r` for reverse, `u` for user-defined.

Function	Sorts by	Order	Keys
`sort()` / `rsort()`	value	ascending / descending	**discarded**, renumbered from 0
`asort()` / `arsort()`	value	ascending / descending	preserved
`ksort()` / `krsort()`	key	ascending / descending	preserved
`usort()` / `uasort()`	your function	your rule	discarded / preserved

> **IF THE KEYS MEAN SOMETHING, NEVER CALL `sort()`.** It sorts the grades and destroys every name, renumbering from 0.

***Table 3.5**: `asort()` sorts by value and keeps the names; `ksort()` sorts by name. Book: Sections 3.13.4–3.13.6.*

Strings

Double quotes interpolate and honor `\n`; single quotes do neither; `"{$name}'s laundry"`; heredoc and nowdoc for many lines.

`mb_strlen()` counts characters, `strlen()` bytes; `trim()`, `str_contains()`, `substr()`, `str_replace()`, `sprintf()`.

`number_format(24500, 0, ',', '.')` is `24.500`: Indonesian puts a dot between thousands.

> **POSITION 0 IS FALSY.** `if (strpos($s, $n))` is false both when the needle is absent and when it comes first. Write `!== false`, or use `str_contains()`.

Java's `indexOf()` and Python's `find()` return `-1` instead, and so do not have this trap. Book: Section 3.14.

Splitting a Program Across Files

Table 3.6 The four including statements. Use `require_once` for anything your program cannot run without, which is nearly everything.

Statement	If the file is missing	If already loaded
`include`	warning, execution continues	loads it again
`require`	fatal error, execution stops	loads it again
`include_once`	warning, execution continues	does nothing
`require_once`	fatal error, execution stops	does nothing

Listing 3.4 From `lib/money.php`: rupiah formatting and the price rule BR-1, price = unit price × units.

```
/** 32000 -> "Rp 32.000": Indonesian digit grouping, no decimals. */
function rupiah(int $amount): string {
    return 'Rp ' . number_format($amount, 0, ',', '.');
}

/** Rule BR-1 of the finished application: price = unit price x units. */
function price(int $unitPrice, int $units): int {
    return $unitPrice * $units;
}
```

Table 3.6*, *Listing 3.4: *three suits (S0005) cost* 3 × 65,000 = 195,000, `Rp 195.000`. *Book: Section 3.16.*

The Superglobals

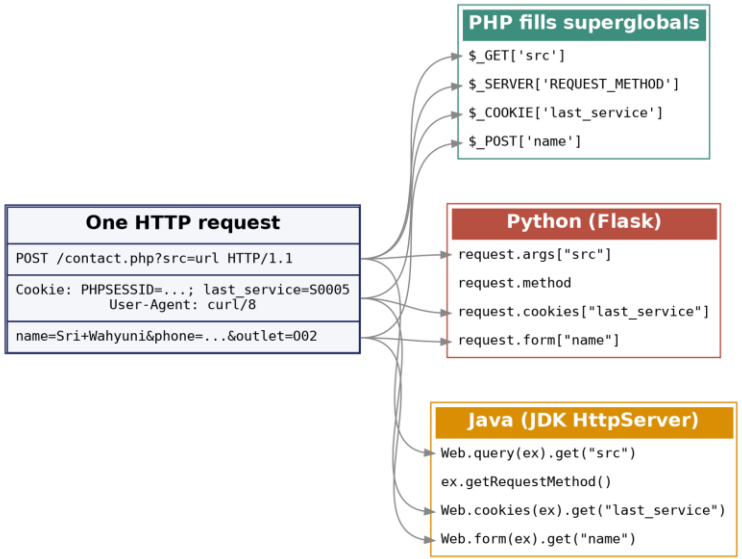


Figure 3.6 One HTTP request, read three ways. The request line fills `$_GET` and `$_SERVER`, the `Cookie` header fills `$_COOKIE`, and the body fills `$_POST`; Java and Flask expose the same parts under other names.

***Figure 3.6**: the request line fills `$_GET` and `$_SERVER`, the `Cookie` header `$_COOKIE`, the body `$_POST`; Java and Flask expose the same parts under other names. Book: Section 3.17.1.*

What Each One Holds

Table 3.7 The superglobals and where each is used in this chapter.

Superglobal	Holds
`$_GET`	values from the URL's query string
`$_POST`	values from the body of a POST request
`$_COOKIE`	cookies the browser sent back (Section 3.20)
`$_SESSION`	data kept for this visitor between requests (Section 3.20)
`$_FILES`	uploaded files (Section 3.22)
`$_SERVER`	facts about the request and the server
`$_REQUEST`	`$_GET` and `$_POST` merged (the `request_order` setting decides)
`$_ENV` / `$GLOBALS`	environment variables / every global variable

> **AVOID `$_REQUEST`.** It merges GET and POST without saying which, so an attacker can supply a value just by sending someone a link.

***Table 3.7**: a **superglobal** is a predefined associative array, filled before our code runs, visible everywhere. Book: Section 3.17.1.*

Everything Arrives as a String

```
curl -s -A 'curl-demo/1.0' "http://127.0.0.1:8080/09_superglobals.php?name=Sri&units=3"
```

`?` starts the query string, `&` separates the pairs: `$_GET['name']` is `"Sri"` and `$_GET['units']` is the **string** `"3"`.

A POST body fills `$_POST`, a `-b` cookie `$_COOKIE`; `Sri+Wahyuni` arrives decoded as `Sri Wahyuni`.

Java: `Web.query(ex)`, `Web.form(ex)`, `Web.cookies(ex)` in a helper written once; Flask: `request.args`, `request.form`, `request.cookies`.

Three programs, one request, the same four facts: the method, the query, the form and the cookie. Book: Sections 3.17.2 and 3.17.3.

GET Asks, POST Does

Table 3.8 GET against POST. The deciding question is not “is it secret” but “does sending it twice do something twice”.

	GET	POST
Data travels in	the URL	the request body
Visible in the address bar, history, server logs	yes	no
Can be bookmarked or shared	yes	no
Size	limited in practice to a few thousand characters	effectively unlimited
Can send files	no	yes, with `enctype="multipart/form-data"`
Safe to repeat	should be	should not be
Use it for	searches, filters, page numbers, calculations	creating, changing, deleting, logging in, logging out

***Table 3.8**: the calculator is a GET, because refreshing a price is harmless; the contact form is a POST. POST is not encrypted — only HTTPS encrypts. Book: Section 3.17.4.*

Forms, and the Helpers

> **KEY INSIGHT.** `<input name="units">` arrives as `$_GET['units']`, `Web.query(ex).get("units")`, `request.args["units"]`. No `name`, no key.

Listing 3.5 From `lib/validate.php`: `field()` reads one value safely, `h()` escapes for HTML.

```
/** One field of $_GET, $_POST or $_COOKIE, always a trimmed string ('' if absent). */
function field(array $src, string $key): string {
    $v = $src[$key] ?? '';
    return is_string($v) ? trim($v) : '';     // "name[]=x" arrives as an array: refuse it
}

/** Escape for HTML text and attribute values. Use it on EVERY value you print. */
function h(?string $s): string {
    return htmlspecialchars((string) $s, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
}
```

***Listing 3.5**: an unticked checkbox and an unchosen radio send nothing; `name="extras[]"` makes PHP deliver an array, which `field()` refuses. `whole()` accepts `^\d{1,9}$` and a range, nothing else. Book: Sections 3.18.1 and 3.18.2.*

The Price Calculator: Read, Validate

Listing 3.6 From `calculator.php`: copy the input into `$old`, validate, and compute the result from the server's own price list.

```
if (isset($_GET['service'])) {                        // the form was submitted
    $old = ['service' => field($_GET, 'service'), 'units' => field($_GET, 'units')];
    [$clean, $errors] = validateCalculation($_GET);
    if (!$errors) {
        $s = SERVICES[$clean['service']];
        $result = [
            'description' => $s['description'],
            'units'       => $clean['units'],
            'unit'        => unitName($s['description'], $clean['units']),
            'price'       => $s['price'],
            'total'       => price($s['price'], $clean['units']),
        ];
```

***Listing 3.6**: the visitor chooses which service; the server decides what it costs. Copying into `$old` is the step beginners leave out, which is why forms empty themselves after one mistake. Book: Section 3.18.3.*

Worked Example: Three Requests

`?service=S0005&units=3`: `3 items x Rp 65.000 = Rp 195.000`, and `Set-Cookie: last_service=S0005; … HttpOnly; SameSite=Lax`.

`?service=S0001&units=4`: `4 kg x Rp 8.000 = Rp 32.000`, with `Wash, dry & fold` escaped by `h()`.

`?service=S0099&units=3.5`: two messages, and the visitor's `3.5` still in the box.

```
  <p class="err">Choose one of the services on the list.</p>
  <input id="units" name="units" value="3.5" inputmode="numeric">
  <p class="err">Enter a whole number from 1 to 999; round kg up.</p>
```

Java's `LaundrySite` and Flask's `app.py` answered the same three requests with the same lines: three languages, one answer, Rp 195.000. Book: Sections 3.18.3–3.18.5.

The Calculator in Flask

Listing 3.7 From `app.py`: the calculator in Flask; `response.set_cookie` writes the same header as `setcookie()`.

```
@app.route("/calculator")                          # GET only: working out a price only asks
def calculator():
    errors, result = {}, None
    old = {"service": shop.field(request.cookies, "last_service"), "units": ""}
    save_cookie = None
    if "service" in request.args:                  # the form was submitted
        old = {"service": shop.field(request.args, "service"), "units": shop.field(request.args, "units")}
        clean, errors = shop.validate_calculation(request.args)
        if not errors:
            s = shop.SERVICES[clean["service"]]
            result = {"description": s["description"], "units": clean["units"],
                      "unit": shop.unit_name(s["description"], clean["units"]),
                      "price": s["price"], "total": shop.price(s["price"], clean["units"])}
            save_cookie = clean["service"]
    page = render_template("calculator.html", services=shop.SERVICES, old=old, errors=errors, result=result)
    response = app.make_response(page)
    if save_cookie:
        response.set_cookie("last_service", save_cookie, max_age=30 * 24 * 3600, httponly=True, samesite="Lax")
    return response
```

***Listing 3.7**: Jinja escapes every `{{ … }}` by itself. Book: Section 3.18.5.*

The Hole, Demonstrated

Listing 3.8 `unsafe.php`, deliberately broken. It works for every visitor who behaves, which is what makes it dangerous.

```php
<?php
// unsafe.php — DELIBERATELY BROKEN. This is the counter-example in the book.
// It echoes whatever arrives without checking or escaping anything.
// NEVER write a page like this. Run it once, see the hole, then close it.
?>
<!DOCTYPE html>
<html lang="en"><head><meta charset="utf-8"><title>Unsafe greeting</title></head>
<body>
<p>Hello, <?php echo $_GET['name'] ?? 'stranger'; ?>!</p>
</body></html>
```

```
<p>Hello, <script>alert("stolen")</script>!</p>
```

***Listing 3.8**: the visitor supplied text, the program treated it as content, the browser as code. This is **cross-site scripting** (XSS); stored and shown to others, every reader runs it. Book: Section 3.19.1.*

The Repair

Listing 3.9 `safe.php`: the only change is `h(…)` around the value.

```php
<?php
// safe.php — the same page, one function away from being correct.
require_once __DIR__ . '/lib/validate.php';
?>
<!DOCTYPE html>
<html lang="en"><head><meta charset="utf-8"><title>Safe greeting</title></head>
<body>
<p>Hello, <?= h($_GET['name'] ?? 'stranger') ?>!</p>
</body></html>
```

```
<p>Hello, &lt;script&gt;alert(&quot;stolen&quot;)&lt;/script&gt;!</p>
```

> **KEY INSIGHT.** Escape on **output**, at the moment of printing, for the destination you print into — every value, every time.

***Listing 3.9**: `htmlspecialchars()` turns `&`, `<`, `>`, `"` and `'` into entities, and those five characters are the whole mechanism by which text becomes markup. Book: Section 3.19.2.*

Validation Is Not Escaping

> **DEFINITION. Validation** decides whether a value is acceptable *at all*, and refuses it if not. **Escaping** makes an accepted value safe for *one particular destination*.

Table 3.9 One escaping rule per destination. The wrong one is the same as none: `htmlspecialchars()` does nothing to protect an SQL query.

Destination	PHP	Java	Python	Chapter
HTML text or attribute	`htmlspecialchars()` with `ENT_QUOTES`	`Web.h()`	Jinja `{{ }}`, `html.escape()`	this one
A URL or query string	`urlencode()`	`URLEncoder.encode()`	`urllib.parse.quote()`	this one
An SQL query	a **prepared statement**	`PreparedStatement`	`?` placeholders	5
JavaScript	`json_encode()`	a JSON library	`json.dumps()`	later

***Table 3.9**: `' OR '1'='1` in a glued SQL string is SQL injection, answered in Chapter 5 by a prepared statement. Every injection flaw is one sentence: data was allowed to become code. Book: Section 3.19.3.*

Six Decisions, Two Walls

1. The rest of the program uses only `$clean`. **2.** Collect every error, not just the first.

3. Check the outlet against the list itself: an **allowlist** beats a blocklist. **4.** Clean the phone, then judge it: `^\d{8,15}$`.

5. Count characters, not bytes: `mb_strlen()`. **6.** Names allow letters of every alphabet, `\p{L}\p{M}`, and `'`, `.`, `-`, space.

```
<input id="name" name="name" value="&lt;script&gt;alert(&quot;stolen&quot;)&lt;/script&gt;">
<p class="err">Use letters, spaces, apostrophes, dots and hyphens only.</p>
```

*Validation refused the value and escaping made the redisplay harmless: either would have been enough, and having both is **defense in depth**. Browser checks are a courtesy; server checks are the fence. Book: Section 3.19.4.*

Cookies: a Header Each Way

```
response:  Set-Cookie: last_service=S0005; Max-Age=2592000; Path=/; HttpOnly; SameSite=Lax
requests:  Cookie: last_service=S0005
```

`Max-Age` is seconds: $30 \times 24 \times 3600 = 2{,}592{,}000$. `HttpOnly` hides it from scripts; `SameSite=Lax` keeps it off other sites' forms; add `Secure` on HTTPS.

A cookie is a header, so it goes **before any output**; and it is visitor input, as untrusted as `$_GET`.

HTTP is stateless and PHP forgets every variable, yet a shop needs memory; `setcookie()`, `Web.setCookie()` and `response.set_cookie()` write the same header. Book: Sections 3.20.1 and 3.20.2.

A Session, Step by Step

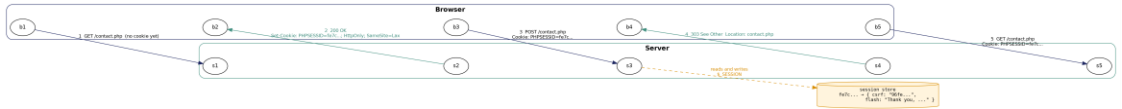


Figure 3.7 A session, step by step. The browser holds only the id `PHPSESSID`; the data lives on the server. Steps 3 to 6 are also the post/redirect/get pattern of Section 3.20.5.

Figure 3.7*: the browser holds only a random id, `PHPSESSID`; the data lives on the server, and `session_start()` loads it into `$_SESSION`. Book: Section 3.20.3.*

Cookie or Session?

Table 3.10 Cookies against sessions. A session is built on a cookie, but only the id travels.

	Cookie	Session
The data lives	in the browser	on the server
The browser holds	the data itself	only a random id
The visitor can read or change it	yes	no (only the id)
Size	about 4 KB	as much as the server allows
Lifetime	`Max-Age`, or until the browser closes	until it expires or is destroyed
Use it for	preferences: last service, language	who is logged in, form tokens, one-time messages
PHP / Java / Flask	`$_COOKIE`, `setcookie()` / `Web.cookies()`, `Web.setCookie()` / `request.cookies`, `set_cookie()`	`$_SESSION` / `Web.session()` / `session`

***Table 3.10**: a login cannot live in a cookie the visitor can edit. Flask's `session` is different in kind — signed, not encrypted, stored in the cookie itself — so never put a secret in it. Book: Sections 3.20.3 and 3.20.6.*

The Form Token and the One-Time Message

Listing 3.10 From `lib/session.php`: the CSRF check and the one-time “flash” message.

```
/** True if the form sent back this session's token. */
function csrfOk(array $post): bool {
    $token = $_SESSION['csrf'] ?? '';
    return $token !== '' && hash_equals($token, field($post, 'csrf'));   // '' never matches
}

/** Keep a message for the NEXT page only (post/redirect/get). */
function flash(string $message): void {
    $_SESSION['flash'] = $message;
}

/** Take the waiting message, if there is one, and forget it. */
function takeFlash(): ?string {
    $m = $_SESSION['flash'] ?? null;
    unset($_SESSION['flash']);
    return $m;
}
```

***Listing 3.10**: another site can make the browser send the cookie but cannot read the token. `hash_equals()` takes constant time, and `$token !== ''` stops an empty token matching an empty field — a bug caught while writing the Java version. Book: Sections 3.20.3 and 3.20.4.*

Post, Redirect, Get

Listing 3.11 From `contact.php`: validate, store, remember a thank-you in the session, and answer `303 See Other`.

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    foreach (array_keys($old) as $k) {
        $old[$k] = field($_POST, $k);                // so the form can be drawn again, filled in
    }
    if (!csrfOk($_POST)) {
        $errors['form'] = 'This form has expired. Please send it again.';
    } else {
        [$clean, $errors] = validateContact($_POST);
    }
    if (!$errors) {
        saveMessage($clean);
        flash('Thank you, ' . $clean['name'] . '. ' . OUTLETS[$clean['outlet']] . ' will call you back.');
        header('Location: contact.php', true, 303);   // post/redirect/get
        exit;
    }
}
```

***Listing 3.11**: after a redirect, Refresh repeats only the harmless GET. Real run: four bad fields give four messages; no token gives "This form has expired"; a good post gives `303` and `Location: contact.php`; the thank-you shows once. Book: Section 3.20.5.*

Never Store a Password

> **DEFINITION.** A **password hash** is the output of a deliberately slow one-way function that mixes in a random **salt**. From the hash you cannot compute the password, only test guesses, one slow guess at a time.

Listing 3.12 From `make_hash.php`: `password_hash()` and `password_verify()`.

```
$hash = password_hash($password, PASSWORD_DEFAULT);        // bcrypt today, with a random salt
echo $hash, PHP_EOL;
echo 'length: ', strlen($hash), PHP_EOL;
echo 'algorithm: ', password_get_info($hash)['algoName'], PHP_EOL;
echo 'verify right password: ', var_export(password_verify($password, $hash), true), PHP_EOL;
echo 'verify wrong password: ', var_export(password_verify('kasir#2026', $hash), true), PHP_EOL;

$again = password_hash($password, PASSWORD_DEFAULT);       // the same password once more ...
echo 'same hash twice? ', var_export($hash === $again, true), PHP_EOL;   // ... a different salt
```

```
length: 60
algorithm: bcrypt
verify right password: true
verify wrong password: false
same hash twice? false
```

***Listing 3.12**: `$2y$12$...` names bcrypt at cost 12, then salt and hash; the same password never gives the same hash. SHA-256 alone is not a password hash: use Argon2id, scrypt, bcrypt or PBKDF2. Book: Sections 3.21.1 and 3.21.2.*

The Staff Login

Listing 3.13 From `login.php`: check the token, verify against the hash, and give the session a new id.

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $email = field($_POST, 'email');
    $password = $_POST['password'] ?? '';            // a password is NOT trimmed
    if (!csrfOk($_POST)) {
        $error = 'This form has expired. Please send it again.';
    } elseif (is_string($password) && checkLogin($email, $password)) {
        session_regenerate_id(true);                 // a new session id at every login
        $_SESSION['staff'] = strtolower($email);
        header('Location: messages.php', true, 303);
        exit;
    } else {
        $error = 'E-mail or password is wrong.';     // never say which of the two
    }
}
```

***Listing 3.13**: one message for a wrong password and an unknown e-mail; the password is **not trimmed**; `session_regenerate_id(true)` defeats session fixation; an unknown e-mail still runs a dummy `password_verify()` so timing reveals nothing. Book: Section 3.21.3.*

The Same Login in Java

Listing 3.14 From `Passwords.java`: PBKDF2 verification in the finished application's format, compared in constant time.

```
    /** password_verify(), for this format. */
    static boolean verify(String password, String stored) {
        String[] p = stored.split("\\$");
        if (p.length != 4 || !p[0].equals("pbkdf2_sha256")) {
            return false;
        }
        byte[] salt = Base64.getDecoder().decode(p[2]);
        byte[] expected = Base64.getDecoder().decode(p[3]);
        byte[] key = pbkdf2(password, salt, Integer.parseInt(p[1]), expected.length);
        return MessageDigest.isEqual(expected, key);            // compares in constant time
    }
```

Stored formats: PHP bcrypt `$2y$12$…` (60 characters), Java `pbkdf2_sha256$600000$…` (90), Flask's Werkzeug `scrypt:32768:8:1$…` (162).

***Listing 3.14**: the finished application stores one PBKDF2 format that all its tracks understand — a design decision, not a language feature. Book: Section 3.21.4.*

Accepting a File

Listing 3.15 From `upload.php`: the real type comes from the bytes, and the name is invented.

```
        // Never trust $f['type'] — the browser sends it and a browser can lie.
        $finfo = new finfo(FILEINFO_MIME_TYPE);
        $mime  = $finfo->file($f['tmp_name']);
        if (!isset(ALLOWED[$mime])) {
            $error = 'Only JPEG, PNG or PDF is accepted (this file is ' . $mime . ').';
        } else {
            // Never trust $f['name'] either — invent our own name.
            $safe = 'slip-' . bin2hex(random_bytes(8)) . '.' . ALLOWED[$mime];
            if (!is_dir(UPLOAD_DIR)) { mkdir(UPLOAD_DIR, 0775, true); }
            if (move_uploaded_file($f['tmp_name'], UPLOAD_DIR . '/' . $safe)) {
                $message = 'Stored as ' . $safe . ' (' . $f['size'] . ' bytes, ' . $mime . ').';
```

```
<p class="ok">Stored as slip-5cfde896d7d01c46.png (67 bytes, image/png).</p>
<p class="err">Only JPEG, PNG or PDF is accepted (this file is text/x-php).</p>
```

***Listing 3.15**: `enctype="multipart/form-data"`; test `error` first; `is_uploaded_file()` defeats a forged path; the extension comes from an allowlist. The browser said `image/png`, the bytes said PHP, and the bytes won. Book: Section 3.22.*

Reading Errors

```
Parse error: syntax error, unexpected token "echo", expecting "," or ";" in
/home/claude/kit/WebPro/Code_new/Chapter03/php/errors/parse_error.php on line 4
```

Parse error: nothing ran. When it names line n, look at line $n - 1$ — here the missing `;` after `echo "one"`.

Warning: "something is wrong and I carried on" — a bug that has not been noticed yet.

Fatal error: "I cannot continue"; read the stack trace from the bottom up. A `TypeError` from `strict_types` is the error you *want*.

"Headers already sent": something was printed before `header()`, `setcookie()` or `session_start()`.

`display_errors` on while developing and off in production; `log_errors` on always; `error_reporting` at `E_ALL`. Book: Section 3.23.

One Idea, Three Stacks

Table 3.11 The chapter's main ideas in three stacks. Learn the row, and the column you happen to sit in stops mattering.

Idea	PHP	Java (JDK HttpServer)	Flask
A URL finds its code through	a file on disk	`createContext()` and a handler	`@app.route` and a function
Query value	`$_GET['q']`	`Web.query(ex).get("q")`	`request.args.get("q")`
Form field	`$_POST['name']`	`Web.form(ex).get("name")`	`request.form.get("name")`
Cookie in / out	`$_COOKIE`, `setcookie()`	`Web.cookies()`, `Web.setCookie()`	`request.cookies`, `set_cookie()`
Session	`session_start()`, `$_SESSION`	`Web.session(ex)` (hand-built)	`session` (signed cookie)
Escape for HTML	`htmlspecialchars()` in `h()`	`Web.h()`	automatic in Jinja
Redirect after POST	`header('Location: …', true, 303)`	`Web.redirect(ex, …)`	`redirect(…, code=303)`
Password hash	`password_hash()` / `password_verify()`	PBKDF2 in `Passwords.java`	Werkzeug

***Table 3.11**: learn the row, and the column stops mattering. Book: Section 3.24.*

Profitina in Practice

Every input is a fence: a website address a brand owner types is parsed, shaped, checked against an allowlist — the shape of `validateContact()`.

Escaping at output, every time: the community feed shows one user's text to every other user.

Sessions and passwords by the book: Argon2id hashes, and a rate-limited login.

Personal data is law: under UU PDP 27/2022, explicit consent, export, password change and full deletion.

What carries across is the shape: untrusted input, a validator, an escaper, a session, a hash. The runnable case is always Profitina Laundry. Book: Section 3.25.

Common Mistakes

`=` for `==`; reading `$_POST` on a GET; trusting browser validation or a cookie.

Printing input without `h()`; answering a successful POST with a page instead of a 303.

A form token that `''` can match; storing a password or a fast hash of it; trimming a password.

Keeping the session id across a login; trusting `$_FILES['x']['type']` or `['name']`.

Output before `header()`, `setcookie()` or `session_start()`: leave the closing `?>` off pure-PHP files.

Ten mistakes are listed in the book, each with its fix. Book: Section 3.26.

Summary: Five Things to Remember

1. A server-side language runs on the server, and its output is the page (**Figure 3.1**); each request is answered on its own.

2. PHP juggles types, so compare with `===`; `foreach` for arrays; functions take copies unless a parameter has `&`; `sort()` destroys string keys.

3. The superglobals bring the request in, and every value is a string from a stranger. GET asks; POST does, and a successful POST redirects.

4. Validate on input, escape on output (**Listing 3.8** against **Listing 3.9**).

5. Cookies in the browser, sessions on the server; tokens on forms; a new session id at login; store only slow, salted password hashes.

Read the OWASP Top 10 once this year, and its cheat sheets on XSS and password storage before deploying anything. Book: Section 3.27.

Next Time

Chapter 4 — Evaluation and Discussion I. Eleven problems on Chapters 1 to 3, from a five-page personal site to a validated, escaped drop-off form.

Every problem worked in full: understood by hand, solved in the stack it names and in Java and Python, and then a tempting wrong solution.

Before then: work Exercises 3.1, 3.2, 3.9, 3.14, 3.15 and 3.16, and run all three tracks of the Laundry pages.

Chapter 4 exercises every idea of today's three sections on the wire, in the browser and on the server. Book: Sections 3.28 and 3.29.

Lecture 4 — Evaluation and Discussion I: From a Static Site to a Validated Form

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 4

Lecturer: Irfan Subakti — yifana@gmail.com

No new language feature today: eleven complete problems on Chapters 1 to 3, each solved from nothing, checked by a program, and followed by a tempting wrong solution. Book: Chapter 4, Sections 4.1–4.17.

What This Meeting Covers

1. **A method** — eight steps for any small web problem, and three instruments that show what really happened.
2. **Problems 4.1–4.4, one project** — a five-page personal site, clean URLs, a report built on evidence, and publication with its source.
3. **Problems 4.5–4.8, the wire and the page** — one status request byte by byte, GET or POST, the price list as a real table, the cascade and the box model by hand.
4. **Problems 4.9–4.11, the numbers and the form** — an exact live estimate, a bill priced in PHP, and a drop-off form validated and escaped.
5. **The problems side by side**, and the traps they share.

Every problem is solved in the technology it asks for and again in Java and Python; every output quoted is real. Book: Section 4.1.

Where We Stand: the Toolkit

Table 4.1 The eleven problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most.

Problem	Topic	Main tools from Chapters 1–3
4.1	Five pages of a personal website	HTML structure, one stylesheet, one script
4.2	Clean URLs for the five pages	paths, the index document rule, 301 and 404, a PHP router
4.3	The project report	measuring a site, the developer tools, writing evidence
4.4	Deploying and publishing the source	Git, a static host, checking the live site
4.5	One request on the wire	URL parts, request and response, status codes
4.6	GET or POST	query string against body, percent-encoding, redirect after POST
4.7	The price list as a table	semantic HTML, `thead`, `th scope`, escaping `&`
4.8	Which rule wins, and how wide is the box	specificity, source order, the box model
4.9	A live price estimate	events, strings from form fields, exact money
4.10	The bill in PHP	associative arrays, `foreach`, `break`, `continue`, sorting
4.11	The drop-off form, validated	`$_POST`, regular expressions, `htmlspecialchars`, 422

Table 4.1 *is a map: if HTTP still feels abstract, begin with 4.5 and 4.6; if CSS surprises you, 4.8; if PHP arrays are shaky, 4.10. Book: Section 4.1.*

Eight Steps for Any Web Problem

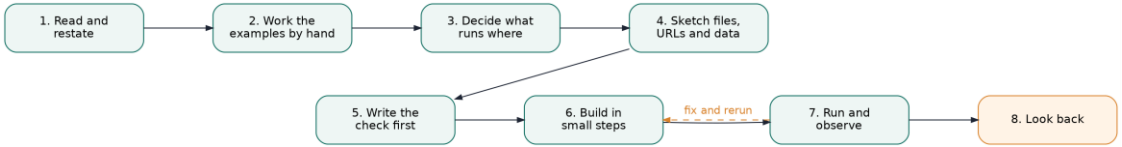


Figure 4.1 The eight steps used for every problem in this chapter; the dashed arrow is the loop between building and observing.

***Figure 4.1**: Pólya's four movements — understand, plan, carry out, look back — stretched into eight steps; the dashed arrow is the loop between building and observing. Book: Section 4.2.*

The Steps, and the One That Matters Most

1 Restate: what goes in, what must come out. **2** Work the examples by hand. **3** Decide what runs where.

4 Sketch files, URLs and data. **5** Write the check first. **6** Build in small steps.

7 Run and observe — the first difference from step 2 *is* the bug. **8** Look back: the tempting wrong way, and a variant.

> **WHY "WHAT RUNS WHERE" COMES FIRST.** Most beginners' web bugs are work placed on the wrong side: a price computed only in JavaScript, a form checked only in HTML, a clean link the server does not know how to answer.

Three instruments: `curl -i` *or* `-v` *shows the conversation in text; the developer tools show it as the browser lives it; a headless browser (Playwright) makes the developer tools a program. Book: Sections 4.2.1–4.2.3.*

Problem 4.1: Five Pages

> **PROBLEM 4.1.** Build a static personal website using HTML, CSS and JavaScript, with a Homepage, a Profile, a Hometown page, a Local Food page and a Tourist Places page.

Static: the server hands back files that already exist; no PHP, no database, no form that sends anything.

Pages: five documents, each with its own title and, in Problem 4.2, its own address — not five sections of one page.

HTML, CSS and JavaScript: one small, honest piece of behavior; complete without the script, a little better with it.

The example is Ayu Pramesti, an Informatics student in Surabaya from Malang: one frame — header, navigation, footer — around five different `main` parts, one `site.css` and one `site.js`. Book: Sections 4.3.1 and 4.3.2.

One Script, Two Small Jobs

Listing 4.1 From `web/portfolio/assets/site.js`: mark the current page, and keep the footer's year current.

```
(function () {
  "use strict";

  // 1. Mark the current page in the navigation, so one nav fragment can be
  //    pasted into all five pages unchanged. Compare normalized paths:
  //    "/portfolio/profile" and "/portfolio/profile/" must count as the same page.
  function normalize(p) {
    return p.replace(/index\.html$/, "").replace(/\/+$/, "") || "/";
  }
  var here = normalize(window.location.pathname);
  document.querySelectorAll("nav.main a").forEach(function (a) {
    if (normalize(a.pathname) === here) {
      a.setAttribute("aria-current", "page");
    }
  });

  // 2. Print the build year in the footer, so the copyright never goes stale.
  var y = document.getElementById("year");
  if (y) { y.textContent = String(new Date().getFullYear()); }
}());
```

***Listing 4.1**: `normalize` makes `/portfolio/food`, `/portfolio/food/` and `…/index.html` one form; `aria-current="page"` is announced by a screen reader and painted by the CSS. Delete the file and the site still works: progressive enhancement. Book: Sections 4.3.3–4.3.5.*

What the Browser Shows

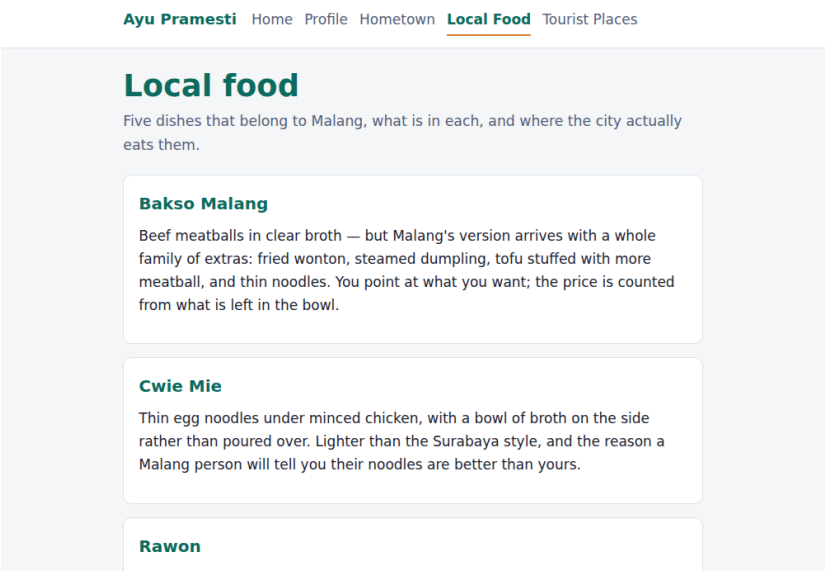


Figure 4.2 The Local Food page at 1000 pixels wide; the script has marked *Local Food* as the current page and the stylesheet has underlined it.

***Figure 4.2**: Local Food is marked as the current page; below 720 pixels the stylesheet, written mobile first, gives one column and a navigation that wraps. Book: Section 4.3.6.*

Java and Flask: the Frame Written Once

Listing 4.2 From `java/PortfolioServer.java`: the menu is written once, and the current page is marked on the server.

```
    static String layout(String slug, String title, String description, String main) {
        StringBuilder nav = new StringBuilder();
        for (Map.Entry<String, String[]> e : PAGES.entrySet()) {
            String current = e.getKey().equals(slug) ? " aria-current=\"page\"" : "";
            nav.append("        <a href=\"").append(href(e.getKey())).append("\"")
               .append(current).append(">").append(e.getValue()[0]).append("</a>\n");
        }
```

In Flask the frame is one Jinja template; `{% if key == slug %} aria-current="page"{% endif %}` does the same job. `curl -s localhost:8402/portfolio/food | sed 's/ aria-current="page"//' | cmp - …/food/index.html` prints nothing: byte for byte the static file.

***Listing 4.2**: a sixth page means one new file and one map entry for Java, against seven edited HTML files for the static site — the whole argument for writing a frame once. Book: Sections 4.3.7 and 4.3.8.*

Tempting and Wrong: One Page, Five Fragments

Five `section`s in one file, shown one at a time by `section:target { display: block; }`: short, and in a browser it looks like five pages.

```
> GET /wrong/onepage.html HTTP/1.1
> Host: localhost:8404
```

The request says `GET /wrong/onepage.html`: `#food` stayed in the browser, so neither the server nor a search engine can tell the “pages” apart.

One title, five `h1`s, and a blank page when opened without a fragment.

A fragment is never sent to the server, so this design can never satisfy Problem 4.2. Book: Section 4.3.9.

Problem 4.2: Clean URLs

> **PROBLEM 4.2.** The pages must answer at `(Your Domain)/portfolio`, `/portfolio/profile`, `/portfolio/hometown`, `/portfolio/food` and `/portfolio/tourist`.

Where does `/portfolio` come from? From a folder of that name — or, on GitHub Pages, from a repository called `portfolio`.

How does an address that names no file find one? The oldest rule of every web server: a request for a **directory** is answered with its `index.html`.

How do links survive the move to a host? By carrying the full prefix.

None of this is HTML: it is decided by the folder layout, the host's configuration, or a program. Book: Sections 4.4.1 and 4.4.2.

Three Fates of One Request

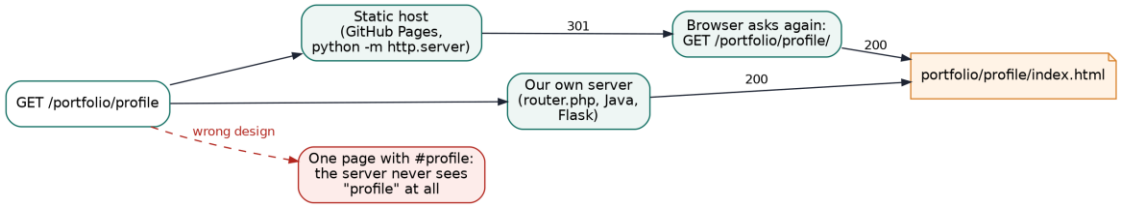


Figure 4.3 Three fates of `GET /portfolio/profile`: a static host redirects to the address with a slash and then sends the index file; our own server sends it at once; a one-page design never lets the server see which page was wanted.

***Figure 4.3**: a static host answers `301` to the slashed address and then `200`; our own server answers `200` at once; a one-page design never lets the server see which page was wanted. Book: Section 4.4.2.*

Which Links Survive?

Listing 4.3 From `web/tools/resolve.mjs`: the browser's own rule, `new URL(href, page)`, on the Profile page of the published site.

```
const page = "https://ayu-pramesti.github.io/portfolio/profile/";
const hrefs = ["/food/", "../food/", "food/", "/portfolio/food/",
               "/assets/site.css", "/portfolio/assets/site.css"];
for (const href of hrefs) {
  console.log(href.padEnd(28) + "-> " + new URL(href, page).href);
}
```

```
/food/                      -> https://ayu-pramesti.github.io/food/
../food/                    -> https://ayu-pramesti.github.io/portfolio/food/
food/                       -> https://ayu-pramesti.github.io/portfolio/profile/food/
/portfolio/food/            -> https://ayu-pramesti.github.io/portfolio/food/
/assets/site.css            -> https://ayu-pramesti.github.io/assets/site.css
/portfolio/assets/site.css  -> https://ayu-pramesti.github.io/portfolio/assets/site.css
```

***Listing 4.3**: `/food/` points above the site, `food/` lands inside the Profile folder, `../food/` is right only from here. Only `/portfolio/food/` is right from every page. Book: Section 4.4.2.*

A Router for `php -S`

Listing 4.4 From `php/router.php`: the host's rules written out; Rule 0 keeps every request inside the `web` folder.

```
// Rule 0: never serve anything outside the web folder ("/../" tricks).
if ($disk !== false && !str_starts_with($disk, $root)) {
    return send_file($root . '/portfolio/404.html', 404);
}
// Rule 1: a real file is sent as it is (stylesheet, script, a page named in full).
if ($disk !== false && is_file($disk)) {
    return send_file($disk, 200);
}
// Rule 2: a directory is answered with its index.html, with or without the slash.
if ($disk !== false && is_dir($disk) && is_file($disk . '/index.html')) {
    return send_file($disk . '/index.html', 200);
}
// Rule 3: nothing matched, so send the site's own error page WITH status 404.
return send_file($root . '/portfolio/404.html', 404);
```

***Listing 4.4**: Rule 2 answers a directory with its `index.html`, slash or no slash, so `/portfolio/profile` gets `200` at once; Rule 3 sends the site's own `404.html` with status 404. Book: Section 4.4.3.*

Four Ways to Clean Addresses

Table 4.2 Four ways to give the five pages clean addresses. The directory layout needs no configuration and works everywhere, which is why the site is built on it; the other three remove the redirect where the host allows it.

Method	Configuration	Works on GitHub Pages	Answer to `/portfolio/profile`
Directory with `index.html`	none	yes	`301`, then `200` at `/portfolio/profile/`
Our own server (PHP router, Java, Flask)	a program	no (Pages runs no program)	`200` at once
Apache `.htaccess`	one file	no	`200` at once
Netlify `_redirects`	one file	no	`200` at once

***Table 4.2**: a rewrite keeps the address in one request; a redirect changes it in two. Netlify writes `_redirects`, Apache reads `.htaccess`; both files are complete, but neither was run for this book. Book: Section 4.4.3.*

Worked Example: Checking All Five

```
PATH                        STATUS  H1 or LOCATION            VERDICT
/portfolio                  200     Ayu Pramesti              ok
/portfolio/profile          200     Profile                   ok
/portfolio/hometown         200     Malang                    ok
/portfolio/food             200     Local food                ok
/portfolio/tourist          200     Tourist places            ok
/portfolio/profile/         200     Profile                   (extra)
/portfolio/assets/site.css  200     -                         (extra)
/portfolio/nonesuch         404     Page not found            (extra)
5 of 5 required URLs reach their page (5 directly, 0 after one redirect)
```

Against the plain static host, every required path answers `301` and then its page: “ok after redirect”. Five out of five both ways.

The tempting mistake is in the links: `href="/profile/"` works at the root of `localhost` and breaks under `/portfolio/`. Test the way the site will be served — serve the parent folder. Book: Sections 4.4.4–4.4.6.

Problem 4.3: a Report on Evidence

> **PROBLEM 4.3.** Write a report with the design concept and structure, screenshots, the code and implementation choices, an analysis or evaluation, and a conclusion.

"The site is fast and has no broken links" is an opinion. "Five pages, 13,354 bytes of HTML; all seven internal addresses answer `200`" is an evaluation, because a reader can check it.

So this writing problem has a programming core: **measure the site**, over HTTP, as a visitor would.

```
PAGE                    STATUS  BYTES  H1  WORDS  DESCRIPTION
/portfolio/             200      2494   1    118  yes
/portfolio/profile/     200      2334   1    102  yes
/portfolio/hometown/    200      2878   1    190  yes
/portfolio/food/        200      2814   1    203  yes
/portfolio/tourist/     200      2834   1    212  yes
total HTML: 13354 bytes in 5 pages
internal links: 7 distinct, 0 broken
```

The same audit in Java, pointed at the Java server, finds every page exactly 20 bytes larger: the length of `aria-current="page"`. A measurement that explains itself to the byte. Book: Sections 4.5.1–4.5.5.

Before and After

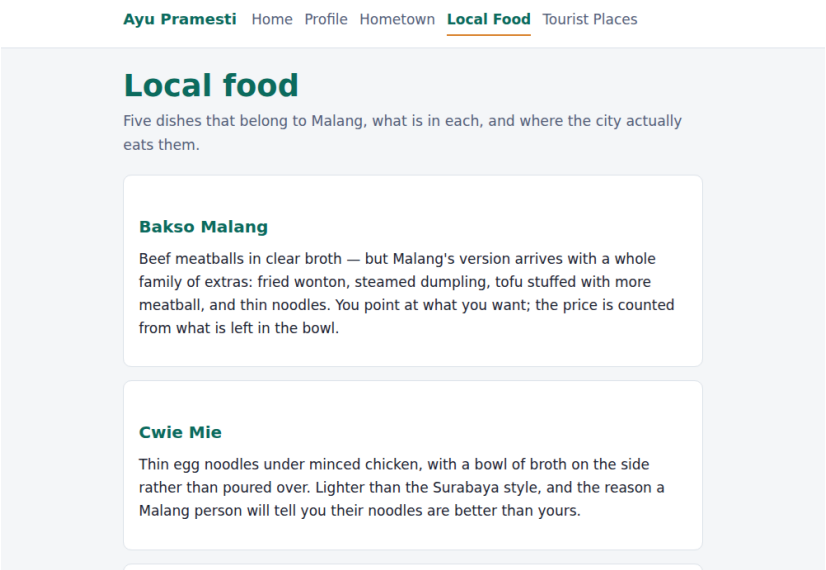


Figure 4.4 The Local Food page before the fix: a gap of 47 pixels between the top of each card and its heading, where the design wanted 19.

***Figure 4.4**: border 1 + padding 18 + the heading's top margin 28 = 47 pixels. After* `.card > :first-child { margin-top: 0; }` *the browser measured 19. Observation, cause with arithmetic, change, measurement: an evaluation paragraph. Book: Section 4.5.6.*

Problem 4.4: Deploy and Publish the Source

> **PROBLEM 4.4.** Upload and deploy the completed website to a free web hosting service, and upload the entire source code to a public GitHub repository.

Listing 4.5 From `web/tools/publish-demo.sh`: commit the site, and push it to a bare repository that stands in for GitHub.

```
git init -q -b main
git -c user.name="Ayu Pramesti" -c user.email="ayu.pramesti@example.ac.id" add .
git -c user.name="Ayu Pramesti" -c user.email="ayu.pramesti@example.ac.id" \
    commit -q -m "Five pages, one stylesheet, one script"
git log --oneline
git ls-files
git init -q --bare -b main "$WORK/github/portfolio.git"
git remote add origin "$WORK/github/portfolio.git"
git push -q -u origin main 2>&1
```

***Listing 4.5**: the repository's root is the site's root; Pages then serves it at `https://ayu-pramesti.github.io/portfolio/`. The rehearsal clones it, serves it on port 8405, and runs the route check. Book: Sections 4.6.1–4.6.3.*

Frozen Pages, and Two Temptations

Pages runs no program, so the Flask site is **frozen**: `app.test_client()` asks for every page, and each answer becomes a file; `/portfolio/nonesuch` becomes `404.html`.

Wrong folder: a repository made from the parent publishes the site one level too deep, `…/portfolio/portfolio/`.

Secrets: a password committed to a public repository stays in its history after it is deleted; list such files in `.gitignore` from the first commit.

> **CHECK YOURSELF.** A typo is fixed in `food/index.html`. `git add food/index.html`, `git commit -m "…"`, `git push`; then `curl -s …/portfolio/food/ | grep -c "the corrected word"` prints 1 once the new version is live.

Static-site generators such as Jekyll are the freezing idea grown up. Book: Sections 4.6.4 and 4.6.5.

Problem 4.5: One Request on the Wire

> **PROBLEM 4.5.** `/status?bill=B2938&phone=0896-0000-1482`: (a) `200` with the status for a known bill and its phone; (b) lower case and spaces or hyphens allowed; (c) `400` without a bill or a phone; (d) `404`, with one message, for an unknown bill *and* for a wrong phone; (e) `405` with `Allow` for other methods.

Use the bills B0120 (0856-0000-8080, collected), B2938 (0896-0000-1482, ready for pickup) and B2946 (0811-0000-8361, in process).

Only the path and the query travel on the request line: the scheme and port decide how to connect, the host goes into the `Host` header, and the fragment never leaves the browser. Book: Sections 4.7.1 and 4.7.2.

Worked by Hand: Six Requests

Table 4.3 The requests of Problem 4.5 worked by hand, with the status and the body each must receive.

Request	Status	Body
`GET /status?bill=B2938&phone=0896-0000-1482`	`200 OK`	`B2938: Ready for pickup`
`GET /status?bill=b2938&phone=0896 0000 1482`	`200 OK`	`B2938: Ready for pickup`
`GET /status?bill=B2938&phone=0811-0000-8361`	`404 Not Found`	`No bill found for this number and phone.`
`GET /status?bill=B9999&phone=0896-0000-1482`	`404 Not Found`	`No bill found for this number and phone.`
`GET /status?bill=B2938`	`400 Bad Request`	`Give both the bill number and the phone number.`
`POST /status?bill=B2938`	`405 Method Not Allowed`	`Use GET to read a status.` and the header `Allow: GET, HEAD`

***Table 4.3**: the answers are decided before any code, and become the check. Book: Section 4.7.2.*

The Status Code Carries the Verdict

Table 4.4 The four status codes of the status check, their meaning in RFC 9110, and why each is the right one here.

Code	Meaning in RFC 9110	Why it fits here
`200 OK`	the request succeeded; the body is the answer	the bill was found and belongs to that phone
`400 Bad Request`	the server will not process the request because of a client error, such as a malformed or incomplete request	a required parameter is missing: nothing can be looked up
`404 Not Found`	the server has no current representation of the target	from the customer's side, "this bill with this phone" does not exist
`405 Method Not Allowed`	the method is not supported by the target; the response must carry an `Allow` header	reading a status is a GET; a POST here is a mistake

***Table 4.4**: a wrong phone gets the same answer as an unknown bill, so typing bill numbers one after another reveals nothing about which bills exist — a personal-data requirement. Book: Section 4.7.2.*

The Status Check in PHP

Listing 4.6 From `php/status.php`: read, clean, look up, and answer with the right code.

```
    $bill  = strtoupper(trim($_GET['bill'] ?? ''));
    $phone = preg_replace('/[\s-]/', '', $_GET['phone'] ?? '');   // "0896-0000 1482" is fine
    $row   = BILLS[$bill] ?? null;
    if ($bill === '' || $phone === '') {
        reply(400, 'Give both the bill number and the phone number.');
    } elseif ($row === null || $row['phone'] !== $phone) {
        // An unknown bill and a wrong phone get the SAME answer: nothing leaks.
        reply(404, 'No bill found for this number and phone.');
    } else {
        reply(200, $bill . ': ' . $row['status']);
    }
```

```
B2938: Ready for pickup
[200]
B2938: Ready for pickup
[200]
No bill found for this number and phone.
[404]
```

***Listing 4.6**: `?? ''` keeps an absent parameter from warning; `strtoupper(trim(…))` accepts `b2938`; `preg_replace('/[\s-]/', '', …)` removes spaces and hyphens. Every line agrees with the table worked by hand. Book: Section 4.7.3.*

Worked Example: the Exact Bytes

```
> GET /status?bill=B2938&phone=0896-0000-1482 HTTP/1.1
> Host: localhost:8411
> Connection: close
>
< HTTP/1.1 200 OK
< Connection: close
< X-Powered-By: PHP/8.4.21
< Content-Type: text/plain; charset=utf-8
<
< B2938: Ready for pickup
```

A request line, the `Host` header HTTP/1.1 requires, `Connection: close`, and the empty line that ends the headers; then a status line, headers, an empty line, the body.

The web of Chapter 1 in the flesh. Java parses `getRawQuery()` itself; Flask returns `(body, status, headers)` and writes `405 METHOD NOT ALLOWED` in capitals — programs read only the number. Book: Sections 4.7.4–4.7.6.

Tempting and Wrong: `200 OK`, "Not Found"

```
HTTP/1.1 200 OK
Connection: close
X-Powered-By: PHP/8.4.21
Content-Type: text/plain; charset=utf-8

No bill found for this number and phone.
```

The body says "not found"; the status line says "found" — and caches, search engines (a **soft 404**) and `fetch` believe the status line.

It also answers `DELETE` like `GET`, and compares the phone exactly, so `0896-0000-1482` as printed on the receipt is "not found".

Check yourself: `?phone=089600001482&bill=B2946&lang=id` gets `200 OK` and `B2946: In process` — `$_GET` is looked up by key, and an unread `lang` is ignored. Book: Section 4.7.7.

Problem 4.6: GET or POST?

PROBLEM 4.6. One form *looks up* a bill; one *creates* a bill from a name, a phone, a service and a quantity. Choose each method; number new bills B3001, B3002, …; make sure a reload never creates a second bill.

Table 4.5 GET and POST compared on the properties that decide which one a form should use.

	GET	POST
Where the form's fields travel	in the query string of the URL	in the body of the request
Visible in the address bar, history, server logs	yes	no (only the URL is)
Can be bookmarked or shared as a link	yes	no
Safe (the client asks for no change)	yes	no
Idempotent (repeating it has the same effect as doing it once)	yes	no
May be repeated by the browser, a proxy or a link preview without asking	yes	no; a browser asks before re-sending
Right for	reading: search, lookup, filter	changing something: create, pay, send

Table 4.5*: the lookup changes nothing, so it is GET and can be bookmarked; the drop-off creates a bill, so it is POST — neither safe nor idempotent. Book: Sections 4.8.1 and 4.8.2.*

Worked Example: 61 Characters, then 303

Siti Aminah drops off 3,5 kg of S0001. A space becomes `+`, a comma `%2C`: 61 characters, so `Content-Length: 61`.

```
0052: Content-Length: 61
0066: Content-Type: application/x-www-form-urlencoded
0000: name=Siti+Aminah&phone=0812-0000-1111&service=S0001&qty=3%2C5
```

Post/Redirect/Get: answer the POST with `303 See Other` and `Location: /orders/B3001`; the browser fetches it with GET, and a reload repeats only the GET.

Listing 4.7 From `php/order.php`: number the bill, append it to the CSV file, and redirect.

```
    $id = sprintf('B%04d', 3001 + count(orders()));   // B3001, B3002, ...
    $fh = fopen(STORE, 'a');
    fputcsv($fh, [$id, ...array_values($fields)], ',', '"', '');
    fclose($fh);
    // Post/Redirect/Get: 303 tells the browser to FETCH the receipt with GET.
    http_response_code(303);
    header('Location: /orders/' . $id);
```

***Listing 4.7**: the POST branch prints no body at all. Fetching the receipt twice created nothing: B3002 does not exist. Java makes `handle synchronized`; Flask writes `redirect(…, code=303)`. Book: Sections 4.8.2–4.8.5.*

Tempting and Wrong: a Link That Creates

```
Created B3001
Created B3002
HTTP/1.1 200 OK
B3001 Siti
B3002 Siti
B3003 Siti
```

Two visits to the link, as Reload would do, and one `curl -I` — a `HEAD` that asked for no content — made three bills for one drop-off.

GET is defined as safe, so restored tabs, chat previews and crawlers may all repeat it.

> **CHECK YOURSELF.** “Mark B2946 as finished” is a POST, answered `303 See Other` with `Location` pointing to the bill.

A GET that changes data is a bug in the server, not in the visitor. Book: Section 4.8.6.

Problem 4.7: the Price List as a Table

> **PROBLEM 4.7.** Twelve services as a complete HTML5 document with a real table — caption, header row, the code as the header of its row, prices with a dot between thousands — and the same document generated by a loop in PHP, Java and Python, byte for byte.

`th scope="row"` and `scope="col"` make 12.000 *the* cell of row S0002 and column *Price (Rp)* — for a screen reader, a program and the CSS.

Two decisions by hand: *Wash, dry & fold* must be written `&`, and 12000 must appear as `12.000`.

The hand-written page is a static file; the three generators run on the server and send exactly that file. Book: Sections 4.9.1 and 4.9.2.

The Rows Written by a Loop

Listing 4.8 From `php/prices.php`: one `foreach` writes the twelve rows, and `e()` escapes every value.

```
    <table id="prices">
      <caption>Services and prices at all three outlets, in rupiah</caption>
      <thead>
        <tr><th scope="col">Code</th><th scope="col">Service</th><th scope="col">Unit</th><th scope="col">Price
(Rp)</th></tr>
      </thead>
      <tbody>
<?php foreach ($services as $code => [$name, $unit, $price]): ?>
        <tr<?= str_starts_with($name, 'Express') ? ' class="express"' : '' ?>><th scope="row"><?= e($code)
?></th><td><?= e($name) ?></td><td><?= e($unit) ?></td><td class="price"><?= number_format($price, 0, ',', '.')
?></td></tr>
<?php endforeach; ?>
      </tbody>
    </table>
```

***Listing 4.8**: `foreach ($services as $code => [$name, $unit, $price])` destructures each entry; `number_format($price, 0, ',', '.')` gives `12.000`. `run_all.sh` compared all three generators with the hand-written file: PHP, Java and Flask output identical. Book: Sections 4.9.3–4.9.5.*

The Price List in a Real Browser



Price list

Per-kg services are weighed at the counter. The minimum charge is 1 kg.

Code	Service	Unit	Price (Rp)
S0001	Wash, dry & fold	kg	8.000
S0002	Express wash & fold, 1 day	kg	12.000
S0003	Ironing only	kg	6.000
S0004	Dry cleaning - shirt/blouse	item	20.000
S0005	Dry cleaning - suit 2 pieces	item	65.000
S0006	Dry cleaning - dress/kebaya	item	45.000
S0007	Dry cleaning - jacket	item	40.000
S0008	Bed cover, any size	item	35.000
S0009	Blanket	item	25.000
S0010	Curtain	kg	15.000
S0011	Shoes deep clean	pair	45.000
S0012	Helmet cleaning	item	30.000

Services and prices at all three outlets, in rupiah

Figure 4.5 The price list in headless Chromium: a note box, a table with a header row and row headers, the express row in red, and the caption below the table.

***Figure 4.5**: a note box, a header row, row headers, the express row in red and the caption below — the colors are Problem 4.8. Book: Section 4.9.3.*

Tempting and Wrong: Cells Without Rows

```
<table id="prices">
    <tbody><tr><th>Code</th><th>Service</th><th>Price</th>
    <td>S0001</td><td>Wash, dry &amp;amp; fold</td><td>8.000</td>
    <td>S0002</td><td>Express wash &amp;amp; fold, 1 day</td><td>12.000</td>
  </tr></tbody></table>
rows: 1
```

The browser does not refuse a table of cells with no `tr`: the HTML standard prescribes a repair, and the repair hides the mistake.

One inserted `tbody`, **one** `tr`, nine cells side by side: `tbody tr:nth-child(2)` matches nothing, and a screen reader announces one row of nine columns.

> **CHECK YOURSELF.** *Carpet < 5 m^2* must appear as `Carpet < 5 m²`: escaping belongs to the moment of output, done by the generator, not by whoever types the data.

`dom_dump.py` asks headless Chromium what tree it built, which is not what was typed. Book: Section 4.9.6.

Problem 4.8: Which Rule Wins?

Without a browser: (a) the color of the price cells of S0001 and S0002; (b) of their other cells; (c) the alignment of the prices; (d) the size of the note box. Then check, and redo (a) without R5.

Listing 4.9 From `web/laundry/prices.css`: the rules that decide color and alignment, numbered R1 to R8.

```
/* R1 */ body            { font: 16px/1.5 system-ui, sans-serif; color: #1d2433; margin: 24px; }
/* R2 */ td              { color: #55607a; }
/* R3 */ .price          { color: #0b6b5e; text-align: right; }
/* R4 */ tr.express td   { color: #b3261e; }
/* R5 */ #prices .price  { color: #1d2433; font-weight: 600; }
/* R6 */ td.price        { color: #d9822b; }
/* R7 */ table           { border-collapse: collapse; }
/* R8 */ th, td          { border: 1px solid #dde3ea; padding: 6px 10px; text-align: left; }
```

***Listing 4.9**: R10 gives the note box `width: 320px`, `padding: 12px 16px` and `border: 4px`. Book: Section 4.10.1.*

Specificity, Compared from the Left

$$(a_1, b_1, c_1) > (a_2, b_2, c_2) \Leftrightarrow a_1 > a_2 \ \vee \ (a_1 = a_2 \wedge b_1 > b_2) \ \vee \ (a_1 = a_2 \wedge b_1 = b_2 \wedge c_1 > c_2) \tag{4.1}$$

Table 4.6 The five rules that set a text color in `prices.css`, with the specificity (a, b, c) of each selector.

Rule	Selector	*a*	*b*	*c*	Color
R2	`td`	0	0	1	`#55607a` gray
R3	`.price`	0	1	0	`#0b6b5e` green
R4	`tr.express td`	0	1	2	`#b3261e` red
R5	`#prices .price`	1	1	0	`#1d2433` near-black
R6	`td.price`	0	1	1	`#d9822b` orange

*Equation (4.1) and **Table 4.6**: a ids, b classes, attributes and pseudo-classes, c types and pseudo-elements; one id beats any number of classes,* (1,0,0) > (0,9,9)*. Order only breaks ties. Book: Section 4.10.2.*

Worked Example: the Cascade by Hand

(a) S0001's price cell matches R2, R3, R5, R6; the highest is R5 (1,1,0): `#1d2433`. S0002's adds R4 (0,1,2), still below: **R5 again**. R6 comes later and loses anyway.

(b) S0001's other cells: only R2, **gray**. S0002's: R4 (0,1,2) beats R2, **red**. The code cells are `th`: they inherit R1.

(c) R3 `.price` (0,1,0) beats R8's `td` (0,0,1) although R8 is later: **right-aligned**.

Without R5: S0001's price turns **orange** (R6 (0,1,1)); S0002's turns **red** (R4 (0,1,2) beats R6 on the third number).

A headless Chromium answered `rgb(29, 36, 51)`, `rgb(179, 38, 30)` and `rgb(85, 96, 122)`: `#1d2433`, `#b3261e`, `#55607a` — every answer right. Book: Sections 4.10.2 and 4.10.3.

Worked Example: How Wide Is the Box?

$$W = w + p_{\text{left}} + p_{\text{right}} + b_{\text{left}} + b_{\text{right}} \tag{4.2}$$

WORKED EXAMPLE. $W = 320 + 16 + 16 + 4 + 4 = 360$ pixels. One line is $16 \times 1.5 = 24$ pixels; the note wraps onto two, so the content is 48 and $H = 48 + 12 + 12 + 4 + 4 = 80$ pixels.

```
note box: 360 x 80 px on screen
table rows: 13 | head rows: 1 | caption: True
```

Equation (4.2) holds for `content-box`; with `box-sizing: border-box` the box occupies exactly 320 pixels and its content gets $320 - 40 = 280$*. The margin is outside the box's own size. Book: Sections 4.10.2 and 4.10.3.*

The Cascade as a Program

Listing 4.10 From `python/specificity.py`: specificity as three counts, and the winner by `(specificity, position)`.

```python
def specificity(selector):
    """(a, b, c) = (ids, classes/attributes/pseudo-classes, elements/pseudo-elements)."""
    a = b = c = 0
    for part in selector.split():                       # "tr.express td" -> 2 compounds
        a += len(re.findall(r"#[\w-]+", part))
        b += len(re.findall(r"\.[\w-]+|\[[^\]]*\]|(?<!:):[\w-]+", part))
        c += len(re.findall(r"^[a-zA-Z][\w-]*|::[\w-]+", part))
    return (a, b, c)


def winner(matching):
    """Highest specificity wins; on a tie, the rule written LATER wins."""
    order = {rule: i for i, (rule, _, _) in enumerate(RULES)}
    return max(matching, key=lambda r: (specificity(r[1]), order[r[0]]))
```

***Listing 4.10**: Python compares tuples exactly as the cascade compares specificities, left to right, so `max(…, key=…)` is the rule. The JavaScript and Java versions print the same seven lines. Book: Section 4.10.4.*

Problem 4.9: a Live Estimate

PROBLEM 4.9. Show an estimate *while the customer types*, exact to the rupiah: a comma or a dot as separator, at most one decimal for kg, whole numbers for items, at least 1 kg, and Rp 10.000 for pick-up and delivery. Then compute it again on the server.

$$T = P \cdot q + D, \qquad q = \begin{cases} \max(q_{\text{typed}}, 1) & \text{for per-kg services} \\ q_{\text{typed}} & \text{for per-item services} \end{cases} \tag{4.3}$$

S0001 `3,5`: 8000 × 3.5 = 28000. S0003 `4,1`: 24600. S0001 `0,5`: 1 kg minimum, 8000. S0002 `2.5` with delivery: 40000. S0008 `1,5`: refused.

Equation (4.3): "while they type" can only be JavaScript, but the browser belongs to the customer, so the charged price is computed again on the server. Book: Sections 4.11.1 and 4.11.2.

Exact Money: Count in Tenths

Listing 4.11 From `web/laundry/estimate.js`: read the quantity as whole tenths, then compute in integers.

```
function toTenths(text) {
  const m = String(text).trim().match(/^(\d{1,2})(?:[.,](\d))?$/);
  if (m === null) return null;
  return Number(m[1]) * 10 + Number(m[2] ?? "0");
}
function estimate(code, quantityText, delivery) {
  const service = SERVICES[code];
  if (service === undefined) return { ok: false, message: "Choose a service." };
  const tenths = toTenths(quantityText);
  if (tenths === null || tenths === 0) {
    return { ok: false, message: "Type a quantity such as 3 or 3,5." };
  }
  if (service.unit !== "kg" && tenths % 10 !== 0) {
    return { ok: false, message: "Items are counted in whole numbers." };
  }
  const billed = service.unit === "kg" ? Math.max(tenths, 10) : tenths;  // at least 1 kg
  // Whole numbers only: price * tenths / 10 is exact because every price ends in 000.
  const total = (service.price * billed) / 10 + (delivery ? DELIVERY : 0);
  return { ok: true, total: total, text: "Rp " + rupiah(total) };
}
```

***Listing 4.11**: `4,1` is 41 tenths, and $6000 \times 41/10 = 24600$ exactly, because every price ends in 000. Java uses `long`, Python `//`. Book: Sections 4.11.2–4.11.4.*

Tests Pass; the Tempting Version Does Not

```
PASS  S0001 "3,5"           Rp 28.000
PASS  S0003 "4,1"           Rp 24.600
PASS  S0001 "0,5"           Rp 8.000
PASS  S0002 "2.5"  +delivery Rp 40.000
PASS  S0008 "2"             Rp 70.000
PASS  S0008 "1,5"           Items are counted in whole numbers.
PASS  S0001 "tiga"          Type a quantity such as 3 or 3,5.
PASS  S9999 "1"             Choose a service.
8 of 8 passed

price * kg        = 24599.999999999996
Math.floor(...)   = 24599
24600 + field     = 2460010000
price * Number('') = 0
```

Floating point gives 24599.999999999996, and `Math.floor` loses a rupiah; `+` on a field's string makes 2,460,010,000; `Number("")` is a free order. One strict pattern and integer arithmetic avoid all three. Book: Sections 4.11.3 and 4.11.5.

Problem 4.10: the Bill in PHP

> **PROBLEM 4.10.** Price each line of one drop-off, skip an unknown code, stop at the end marker, pay as much as possible from the customer's deposit, then list the services dearest first with their codes. C0001 has Rp 52.000 in deposit.

`S0001` 35 tenths: $8000 \times 35/10 = 28000$. `S0008`: 35000. `S0X99`: unknown, skip. `S0011` two pairs: 90000. End marker: stop; `S0009` is never reached.

Total $28000 + 35000 + 90000 = 153000$; from the deposit $\min(52000, 153000) = 52000$; still to pay 101000.

*An unknown code is a mistake in one line, so the loop **skips** it — `continue`. The end marker means there are no more lines, so the loop **stops** — `break`. Book: Sections 4.12.1 and 4.12.2.*

The Loop, and What It Prints

Listing 4.12 From `php/bill.php`: `break` stops at the end marker, `continue` skips an unknown code, `intdiv` keeps the money exact.

```
$total = 0;
foreach ($lines as [$code, $tenths]) {
    if ($code === '') {
        break;                                  // end of the list: stop reading lines
    }
    if (!array_key_exists($code, $services)) {
        echo "skipped unknown code $code\n";
        continue;                               // this line only; go on with the next
    }
    [$name, $price] = $services[$code];
    $amount = line_total($price, $tenths);
    $total += $amount;
    printf("%-5s %-28s %5s x %10s = %11s\n", $code, $name,
           number_format($tenths / 10, 1, ',', '.'), rupiah($price), rupiah($amount));
}
```

```
-----------------------------------------------------------------
Total                                                Rp 153.000
Paid from deposit                                     Rp 52.000
Still to pay                                         Rp 101.000
```

***Listing 4.12**: every number is the one worked by hand; `arsort` then lists the prices high to low, keeping each code. Book: Sections 4.12.3 and 4.12.4.*

Tempting and Wrong: `rsort`

Listing 4.13 `php/wrong_sort.php`: `rsort` puts the prices in order and replaces the codes with 0, 1, 2, and so on.

```php
<?php
// wrong_sort.php: TEMPTING BUT WRONG. rsort() sorts the prices and throws the codes away.
// Run:  php wrong_sort.php
$prices = ['S0001' => 8000, 'S0002' => 12000, 'S0003' => 6000,
           'S0008' => 35000, 'S0009' => 25000, 'S0011' => 45000];
rsort($prices);
print_r($prices);
echo 'Dearest service: ', array_key_first($prices), "\n";
```

```
Array
(
    [0] => 45000
    [1] => 35000
Dearest service: 0
```

***Listing 4.13**: the prices are in order and the codes are gone; "the dearest service" is `0`. Keep keys with `asort`, `arsort`, `ksort`, `krsort`. With `continue` in place of `break`, the blanket would be billed too:* 178000 *in total, Rp 126.000 still to pay. Book: Section 4.12.5.*

Problem 4.11: a Validated, Escaped Form

PROBLEM 4.11. A drop-off form that posts to itself. Name: 2 to 60 letters, spaces, dots, apostrophes, hyphens; phone: 10 to 13 digits from 08; a listed service; a quantity as in Problem 4.9. Errors beside each field, values kept, a status that says “not acceptable”, and everything escaped.

Table 4.7 The drop-off form’s hand-worked cases: two submissions that must be accepted and two that must be refused, with the messages expected.

Name	Phone	Service	Quantity	Expected
`Ma'ruf Saputra`	`0817-0000-8341`	S0002	`2,5`	accepted
`Budi`	`081234567890`	S0001	`3`	accepted
`<script>alert(1)</script>`	`12345`	S0008	`1,5`	three messages: name, phone, quantity
`Budi`	`081234567890`	S0001	`0,0`	one message: quantity

***Table 4.7**: Ma’ruf is why apostrophes are allowed. A form that arrives but is not acceptable gets `422 Unprocessable Content`. The form uses POST, so the program reads `$_POST` and never `$_REQUEST`. Book: Sections 4.13.1 and 4.13.2.*

Validate on Arrival

Listing 4.14 From `php/dropoff.php`: one message per bad field; an empty array means all good.

```
function validate(array $in): array            // returns field => error message
{
    $errors = [];
    if (!preg_match("/^[\p{L} .'-]{2,60}$/u", $in['name'])) {
        $errors['name'] = 'Use 2 to 60 letters; spaces, dots, apostrophes and hyphens are allowed.';
    }
    if (!preg_match('/^08\d{8,11}$/', preg_replace('/[\s-]/', '', $in['phone']))) {
        $errors['phone'] = 'Use an Indonesian mobile number such as 0812-3456-7890.';
    }
    if (!array_key_exists($in['service'], SERVICES)) {
        $errors['service'] = 'Choose one of the listed services.';
    } elseif (!preg_match('/^(\d{1,2})(?:[.,](\d))?$/', $in['qty'], $m)
              || (int) $m[1] * 10 + (int) ($m[2] ?? 0) === 0) {
        $errors['qty'] = 'Type a quantity such as 3 or 3,5.';
    } elseif (SERVICES[$in['service']][1] !== 'kg' && ($m[2] ?? '0') !== '0') {
        $errors['qty'] = 'Items are counted in whole numbers.';
    }
    return $errors;
}
```

***Listing 4.14**: `\p{L}` is a letter in any alphabet, and the `u` flag makes PHP read UTF-8, so Zoë passes. Python's `re` spells the same class `[^\W\d_]`. The script tag of row three comes back as `<script>…`, text that never runs. Book: Section 4.13.3.*

Tempting and Wrong: Validated, Not Escaped

```
    <p><label>Name <input name="name" value="" autofocus onfocus="alert(document.cookie)"></label> Please check
the name.</p>
wrong_dropoff.php  input attributes: name, value, autofocus, onfocus alerts: ['hijacked']
dropoff.php        input attributes: name, value                    alerts: []
```

The attack needs no `<script>`, only a quotation mark: it closes `value` and adds `autofocus onfocus=…`. The name was *rejected* and the attack ran anyway.

`strip_tags` would not help: the attack has no tags. Only escaping the quote, `htmlspecialchars(…, ENT_QUOTES)`, keeps the value inside the attribute — a filter is always one character short. Book: Sections 4.13.4 and 4.13.5.

The Problems Side by Side

Table 4.8 The eleven problems of this chapter, the design idea each one teaches, and the trap its tempting wrong solution demonstrates.

Problem	Design idea	Trap shown by the tempting wrong solution
4.1 Five pages	one frame, one stylesheet, one script	five “pages” in one document, told apart by `#fragment`
4.2 Clean URLs	a directory with `index.html` per page; routes on a server	links beginning with `/` that forget the prefix
4.3 The report	every claim backed by a measurement	adjectives instead of evidence
4.4 Deploy and publish	the repository’s root is the site’s root	a site one folder too deep; secrets in the history
4.5 One request on the wire	the status code carries the verdict	`200 OK` with “not found” in the body
4.6 GET or POST	GET reads, POST changes, then redirect	a link that creates a bill
4.7 The price table	real rows, real headers, escaped `&`	cells without rows, repaired silently
4.8 Cascade and box	specificity first, order only for ties	“the last rule wins”
4.9 Live estimate	integers of tenths; the server recomputes	floating-point kilograms; `+` on a string
4.10 The bill	`continue` skips, `break` stops; `arsort` keeps keys	`rsort` throws the codes away
4.11 Validated form	validate on arrival, escape on output	a rejected value printed back raw

***Table 4.8**: almost every trap appears only at an edge — a prefix, a repeated request, a quotation mark, 4,1 kilograms — and almost every correct answer was observed, not assumed. Book: Section 4.14.*

Profitina in Practice

One address per page, an honest status code: topics behind `#fragments` are one page to a crawler, and `200 OK` meaning "not found" is indexed as content.

Structure that says what things are: a real table, and `schema.org` structured data that states a business and its prices.

Escaping what other people wrote: the community feed, streamed with Server-Sent Events, carries users' text into other people's pages.

Evidence rather than adjectives: every score comes with its measurements. The application advises; the owner decides.

The runnable case study stays Profitina Laundry, whose finished application is in `ProfitinaLaundry/App`. Book: Section 4.15.

Common Mistakes

Links that forget the prefix; pages faked with fragments; opinions where evidence is needed; publishing the wrong folder or a password.

The verdict hidden in the body; changing data with GET.

Trusting the browser to repair markup; reading CSS from top to bottom; forgetting padding and border in a width.

Money in floating point; `+` on a form field; the price computed only in the browser.

Sorting away the keys; validating without escaping.

Fourteen mistakes are listed in the book, each with the problem that shows it. Book: Section 4.16.

Summary: Five Things to Remember

1. Eight steps solve a small web problem; step 3, what runs where, prevents the most expensive mistakes. Observe with `curl`, the developer tools and a headless browser.

2. A directory with an `index.html` gives each page a clean address on any host; links carry the full prefix; a report stands on measurements.

3. The status code carries the verdict; GET reads and may be repeated; POST changes and is answered with `303`.

4. Tables need real rows and headers; the cascade chooses by specificity (Equation (4.1)), then order; a box is content plus padding plus border (Equation (4.2)).

5. Money in whole numbers, recomputed on the server; `continue` skips, `break` stops, `arsort` keeps keys; validate on arrival, escape on output.

The habit of looking with an instrument is the most valuable thing these eleven problems leave. Book: Section 4.17.

Next Time

Chapter 5 — PHP and MySQL, and Getting a Server Running. The Profitina Laundry database, SQL, prepared statements against SQL injection, and a real server.

The pages of Chapters 3 and 4 stop reading constants and CSV files and start reading the `Service`, `Bill` and `Customer` tables.

Before then: work Exercises 4.2, 4.5, 4.6, 4.7 and 4.10, and install XAMPP or MySQL as Section 1.16.5 describes.

Chapter 5 answers the question every problem today postponed: where does the data live? Book: Sections 4.18 and 4.19.

Lecture 5 — PHP and MySQL, and Getting a Server Running

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 5

Lecturer: Irfan Subakti — yifana@gmail.com

Today our pages stop forgetting: a server we control, a memory between requests, a database, and the one safety rule that matters most. Book: Chapter 5, Sections 5.1–5.22.

What This Meeting Covers

1. **A server you control** — Apache, PHP and MySQL; XAMPP, LAMP, the document root, ports, `php.ini`.
2. **Memory between requests** — redirection, cookies, sessions, signing out.
3. **A database and its language** — tables and keys, the seven tables of Profitina Laundry, SQL from `CREATE` to `GROUP BY`, transactions.
4. **PHP meets the database** — `mysqli`, an object wrapper, PDO with prepared statements, CRUD for customers and bills.
5. **SQL injection** — demonstrated with eleven characters, then closed for good; the same code in Java and in Python.

The order follows the dependencies: no database without a server, no safe page without prepared statements. Book: Section 5.1.

Where We Left Off

Chapter 3 gave us PHP: a program whose output is a page, `$_GET` and `$_POST`, validation, and escaping on the way into HTML.

Its weakness: when the script ends, everything it knew is gone. A cashier who signed in on one page is a stranger on the next.

Profitina Laundry is a shop: three outlets in Surabaya, fourteen employees, 256 customers, almost three thousand bills a year.

Which bills at Keputih are still waiting? How much did Rungkut earn in March? Who still owes money? With a database, each is one sentence of SQL.

> **WHAT THIS CHAPTER IS REALLY ABOUT.** Everything from here on is about *state*: facts that outlive the request that created them. A session lasts one visit; a database row lasts until somebody deletes it.

Most hard bugs in web programming turn out to be a confusion between these two kinds of state. Book: Section 5.1.

A “Web Server” Is Three Programs

> **DEFINITION.** The **web server** listens on a port and answers HTTP (Apache, Nginx, IIS). The **language interpreter** runs your code (PHP); it has no port. The **database server** stores data and answers SQL on its own port (MySQL or MariaDB).

Apache answers the telephone, PHP does the thinking, MySQL remembers.

A page that will not load points to Apache; a blank page or visible PHP source points to PHP; “could not connect” or “access denied” points to MySQL.

Knowing which program is which turns an afternoon of staring at a blank page into a minute’s fix. Book: Section 5.2.1.

The Three Programs, Drawn

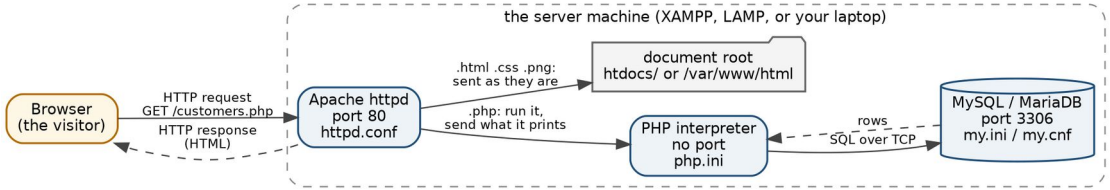


Figure 5.1 The three programs behind every dynamic page: Apache speaks HTTP and serves files, PHP runs the `.php` files Apache hands it, and MySQL or MariaDB stores the data and answers SQL.

In Figure 5.1 the browser never sees PHP or MySQL: it talks only to Apache, which hands `.php` files to PHP, which talks to the database over its own connection. Book: Section 5.2.1.

Ports and Configuration Files

Table 5.1 The three programs, what each does, the port it listens on and the file that configures it. Almost every setup problem is solved by editing one of the three files in the last column.

Program	Job	Default port	Configuration file
Apache `httpd`	speaks HTTP; finds files; passes `.php` files to PHP	80, and 443 for HTTPS	`httpd.conf`
PHP	runs your program	none	`php.ini`
MySQL / MariaDB	stores data, answers SQL	3306	`my.ini` (Windows) or `my.cnf`

Table 5.1: almost every setup problem is solved by editing one of the three files in the last column, and then restarting the program that reads it. Book: Section 5.2.1.

Installing the Stack

Windows: XAMPP into `C:\xampp`; start Apache, then MySQL; check `http://localhost/` and `http://localhost/phpmyadmin/`; add `C:\xampp\php` and `C:\xampp\mysql\bin` to **Path**.

macOS: XAMPP from the `.dmg` and *manager-osx*, or Homebrew: `brew install php mariadb`.

Linux (LAMP): `sudo apt install apache2 php libapache2-mod-php php-mysql php-sqlite3 php-mbstring mariadb-server`, then `sudo systemctl enable --now apache2 mariadb`.

Nothing installed: `php -S localhost:8080` serves a folder, and SQLite keeps the whole database in one file, `laundry.db`.

> **Install one web server, not two.** "Port 80 in use by another application" means IIS, Skype or a second Apache got there first.

XAMPP's MySQL `root` has an empty password: fine on a laptop, unacceptable anywhere else. The document root moves between systems; the three programs, their ports and `http://localhost/` do not. Book: Sections 5.2.2–5.2.5 and 5.2.12.

Apache's Configuration

Listing 5.1 gathers the handful of directives from `httpd.conf` that matter now.

Listing 5.1 `server/httpd-excerpt.conf`: the lines of XAMPP's `httpd.conf` you will actually read. On Ubuntu the same settings live in `/etc/apache2/ports.conf` and `/etc/apache2/sites-available/000-default.conf`.

```
# The lines of XAMPP's httpd.conf (C:\xampp\apache\conf\httpd.conf) that matter now.
Listen 80
ServerName localhost:80
DocumentRoot "C:/xampp/htdocs"
<Directory "C:/xampp/htdocs">
    Options Indexes FollowSymLinks Includes ExecCGI
    AllowOverride All
    Require all granted
</Directory>
DirectoryIndex index.php index.html
```

`Listen` is the port, `DocumentRoot` the folder that becomes the website, `DirectoryIndex` the file sent for a folder. Apache reads the file once, at start: edit, then restart, and read the error log if it refuses. Book: Section 5.2.6.

The Document Root

Table 5.2 The document root turns a folder into a website: the path after the host name is the path inside the folder, and nothing outside the folder has an address at all.

File on disk	Address in the browser
`C:\xampp\htdocs\hello.php`	`http://localhost/hello.php`
`C:\xampp\htdocs\laundry\index.php`	`http://localhost/laundry/index.php`
`C:\xampp\htdocs\laundry\`	`http://localhost/laundry/` (sends `index.php`)
`C:\xampp\php\php.ini`	none: it is outside the document root

WORKED EXAMPLE. A page saved as `C:\xampp\htdocs\profitina\web\customers.php` gives 404. Remove the document root, turn `\` into `/`, prefix `http://localhost/`: **`http://localhost/profitina/web/customers.php`**.

The last row of Table 5.2 is a security property: keep `config.php`, which holds passwords, outside the document root and `require` it from PHP, which reads the disk directly. Book: Section 5.2.6.

Ports, and `php.ini` for Development

Only one program may listen on a port: `netstat -ano | findstr :80` (Windows) or `sudo lsof -i :80` finds the other one; or move Apache to `Listen 8080`.

Listing 5.2 `server/php-development.ini`: the settings for a development machine. Every problem is shown on the page, and the three database extensions this chapter needs are switched on.

```
; php.ini on your own computer: show every problem on the page.
display_errors = On
display_startup_errors = On
error_reporting = E_ALL
date.timezone = Asia/Jakarta
extension=mysqli
extension=pdo_mysql
extension=pdo_sqlite
```

In Listing 5.2, "enabling an extension" means deleting the semicolon in front of it; without `date.timezone`, every Surabaya timestamp is seven hours behind. `php --ini` names the file really read. Book: Sections 5.2.7 and 5.2.8.

Redirection

DEFINITION. A **redirect** is a response with no page, only an instruction: status **302 Found** and a `Location:` header. The browser then makes a second, separate request.

Listing 5.3 `php/state/protected.php`: the guard at the top of every protected page. A visitor with no identity in the session is redirected, and `exit` makes sure nothing below the guard runs for them.

```php
<?php
// protected.php: a page only a signed-in visitor may see.
declare(strict_types=1);
session_start();

if (!isset($_SESSION['user_id'])) {
    header('Location: session.php');     // not signed in: go somewhere else
    exit;                                // and stop here; the rest must not run
}

header('Content-Type: text/plain; charset=utf-8');
echo "Welcome back, ", $_SESSION['user_id'], PHP_EOL;
```

Listing 5.3 shows the two rules: `header()` must run before any output ("headers already sent"), and `exit` must follow, because `header()` only queues a header. Book: Section 5.3.1.

HTTP Has No Memory

> **DEFINITION.** A protocol is **stateless** when each request is handled as a separate event, with no knowledge of any request before it. HTTP is stateless.

A good design: a stateless server can be restarted or multiplied behind a load balancer without anyone noticing.

The memory lives in one of two places. **On the client:** a **cookie**. **On the server:** a **session**, whose ticket number travels in a cookie.

A basket on its way to the checkout, a cashier who signs in once and works for an hour: the application must build that memory itself. Book: Section 5.3.2.

Cookies: Create, Read, Delete

DEFINITION. A **cookie** is a small named piece of text that a server asks a browser to store (`Set-Cookie:`) and to send back with every later request to the same site (`Cookie:`).

Listing 5.4 From `php/state/cookies.php`: `setcookie()` before any output to create, and an expiry in the past to delete.

```
$action = $_GET['action'] ?? 'show';

if ($action === 'set') {
    // setcookie() writes a header, so it must run before any output at all.
    setcookie('hobby', 'swimming', time() + 3600);
    setcookie('last_service', 'S0001', time() + 3600);
}
if ($action === 'delete') {
    // A cookie is deleted by setting it again with a time already past.
    setcookie('hobby', '', time() - 3600);
    setcookie('last_service', '', time() - 3600);
}
```

After Listing 5.4 the page prints `$_COOKIE`, the cookies this request arrived with, as plain text so that the transcript is easy to read. Book: Section 5.3.3.

The Cookie Conversation

```
$ curl -s -i -c jar 'http://localhost:8080/cookies.php?action=set'
HTTP/1.1 200 OK
Host: localhost:8080
Date: Wed, 23 Sep 2026 09:30:49 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Set-Cookie: hobby=swimming; expires=Wed, 23 Sep 2026 10:30:49 GMT; Max-Age=3600
Set-Cookie: last_service=S0001; expires=Wed, 23 Sep 2026 10:30:49 GMT; Max-Age=3600
Content-Type: text/plain; charset=utf-8

action = set
cookies this request arrived with:
  (none)

$ curl -s -b jar http://localhost:8080/cookies.php
action = show
cookies this request arrived with:
  last_service => S0001
  hobby => swimming
```

*The first response sets two cookies and received none: `$_COOKIE` holds what the browser sent, so a new cookie appears on the **next** request. Deleting sets the value `deleted` with an expiry in 1970. Set `HttpOnly`, `Secure` and `SameSite=Lax`. Book: Section 5.3.3.*

What Is Wrong with Cookies

Table 5.3 The four classic problems with cookies. The second is the one that turns into a security incident.

Problem	What it means in practice
The user can switch cookies off	a site that depends on them must say so, or fall back to something else
Other people can see them	a cookie is a file on the visitor's computer; anyone at that computer can read and edit it
They are small and few	browsers historically kept about 20 cookies per site and about 4 kB each
Browsers differ	some browsers only behave correctly when every option of `setcookie()` is given

> **Never put anything in a cookie that you would mind the visitor changing.** Not a price, not a customer id that grants access, not `is_owner=0`.

The second row of Table 5.3 is the one that turns into a security incident. A cookie may hold a harmless preference and a ticket number whose meaning only the server knows. Book: Section 5.3.4.

Cookies as the Visitor Sees Them

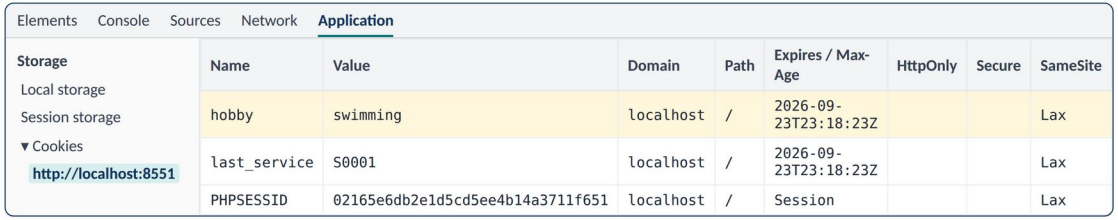


Figure 5.2 The cookies of `http://localhost:8551` in the browser's developer tools (*Application → Storage → Cookies*), after `cookies.php?action=set` and `session.php` ran. Nothing in a cookie is hidden from the person at the computer.

Figure 5.2: F12 shows every name and every value in plain text to anyone at the computer, and each one can be edited. Book: Section 5.3.4.

Sessions

DEFINITION. A **session** is a sequence of requests from one browser that the server treats as belonging together. PHP gives it a **session id**, keeps a private store on the server, and exposes it as `$_SESSION`.

Listing 5.5 `php/state/session.php`: a visit counter kept in the session. The number climbs only if `$_SESSION` really survives from one request to the next.

```php
<?php
// session.php: a visit counter kept in a session instead of a cookie.
declare(strict_types=1);
session_start();

$_SESSION['visits'] = ($_SESSION['visits'] ?? 0) + 1;
if (!isset($_SESSION['user_id'])) {
    $_SESSION['user_id'] = 'owner@profitina-laundry.example';
}

header('Content-Type: text/plain; charset=utf-8');
echo "session id   : ", session_id(), PHP_EOL;
echo "visits       : ", $_SESSION['visits'], PHP_EOL;
echo "user_id      : ", $_SESSION['user_id'], PHP_EOL;
```

In Listing 5.5 the number climbs only if `$_SESSION` really survives from one request to the next. `session_start()` must come before any output. Book: Section 5.3.5.

How a Session Works

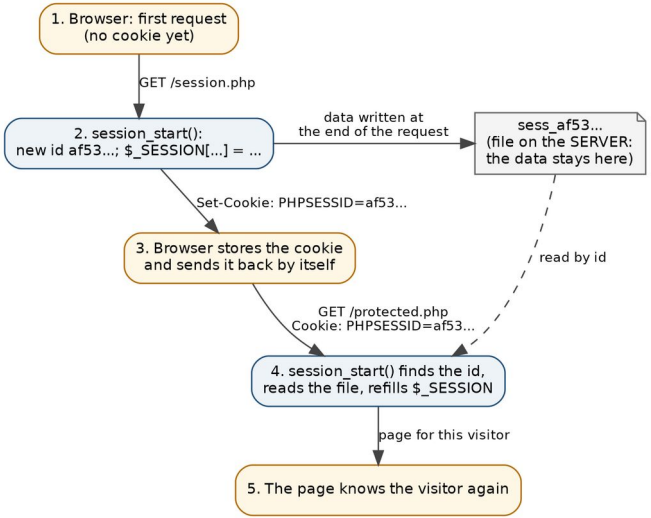


Figure 5.3 How a PHP session works. Only the session id travels to the browser, inside a cookie; the data stays on the server in a store named after that id, and `session_start()` refills `$_SESSION` from it on every later request.

Figure 5.3: `session_start()` *invents an id and sends it in a cookie; the data goes to a file on the server; the next* `session_start()` *refills* `$_SESSION`. ***The data never leaves the server.*** *Book: Section 5.3.6.*

A Session, Watched with `curl`

```
$ curl -s -b jar http://localhost:8080/session.php
session id   : da3299bfacb793dc10089a180f3a8ae0
visits       : 2
user_id      : owner@profitina-laundry.example

$ curl -s -b jar http://localhost:8080/protected.php
Welcome back, owner@profitina-laundry.example

$ curl -s http://localhost:8080/session.php
session id   : e3be2cf6b729fff628f0f00654f68210
visits       : 1
user_id      : owner@profitina-laundry.example
```

The same cookie jar gives the same id and a counter of 2, and reaches the protected page; a request with no jar is a new browser: new id, counter 1. A session belongs to a browser, not to a person. Anyone who learns a session id becomes that visitor. Book: Section 5.3.6.

Signing Out

Listing 5.6 `php/web/logout.php`: signing out properly. The cookie is deleted in the way Section 5.3.3 described, by setting it again with a time in the past.

```php
<?php
// logout.php: empty the session, drop the cookie, destroy the server-side store.
require __DIR__ . '/boot.php';

$_SESSION = [];
$p = session_get_cookie_params();
setcookie(session_name(), '', time() - 3600, $p['path'], $p['domain'], $p['secure'], $p['httponly']);
session_destroy();
header('Location: login.php');
exit;
```

> **COOKIE OR SESSION?** A preference, harmless if edited: a cookie. Anything that says who someone is or what they may do: a session.

Listing 5.6 empties the array, deletes the cookie by setting it with a past time, and destroys the store on the server. Putting the id in the URL (`?PHPSESSID=`) is off in modern PHP, because addresses are copied and logged. Book: Section 5.3.7.

Tables, Rows and Columns

The laundry's counter book is a table: a column for each kind of fact, a row for each bag. A relational database is that book made precise.

> **DEFINITION.** A **table** (*relation*) is a set of rows with the same named columns. A **column** has a name and a **type**. A **row** is one complete set of values, one for each column.

Relational names Codd's 1970 idea: facts are connected *only* by matching values. Bill `B2931` holds `C0056`; another table has `C0056` next to Gilang Nasution.

Rows are a set: they have no order unless you ask with `ORDER BY`. `C0056` is `C0056` on any machine, in any copy, in any year; pointers break when things move. Book: Sections 5.4.1 and 5.4.2.

Keys

DEFINITION. A **primary key** has a different, non-empty value in every row: it names the row. A **foreign key** is a column whose value must be the primary key of a row in another table; the database enforces it.

WORKED EXAMPLE. `Customer.id = 'C0056'` is Gilang Nasution. `Bill.Customer_id` refers to it: bill `B2931` carries `'C0056'`, and that single value is the whole connection. A bill for `C9999` is not merely unfound; it is refused.

The laundry uses surrogate codes such as `C0056` rather than natural keys such as a telephone number, because natural keys change and a changing key drags every foreign key behind it. Book: Section 5.4.3.

The Seven Tables of Profitina Laundry

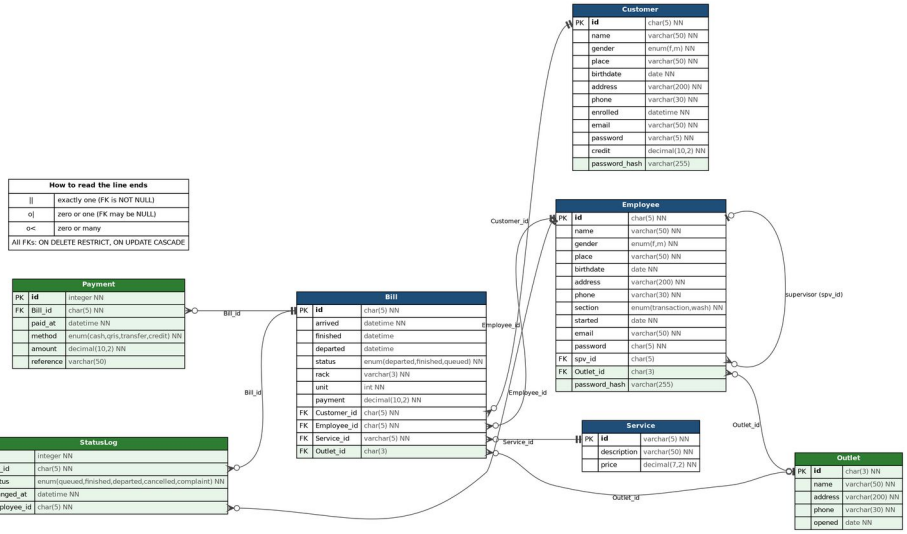


Figure 5.4 The Profitina Laundry schema. Blue tables come from the original design (Customer, Employee, Service, Bill); green tables and columns were added in September 2026 (Outlet, Payment, StatusLog). A line ending in a crow's foot means "many", a double bar "exactly one", and a circle "possibly none".

Figure 5.4: each box a table, each line a foreign key; a crow's foot means "many", a double bar "exactly one", a circle "possibly none". Book: Section 5.4.4.

One Row Is …

Table 5.4 The seven tables in words. The sample data covers one year of the shop, October 2025 to September 2026.

Table	One row is	Primary key	Rows in the sample
`Outlet`	one of the three shops	`id`, e.g. `O02`	3
`Customer`	one registered customer	`id`, e.g. `C0056`	256
`Employee`	one member of staff; `spv_id` names their supervisor	`id`, e.g. `E0004`	14
`Service`	one line of the price list	`id`, e.g. `S0001`	12
`Bill`	one laundry order: one customer, one service, a quantity	`id`, e.g. `B2931`	2,953
`Payment`	one amount of money received for a bill	`id`, a number	3,319
`StatusLog`	one event in a bill's life, and who caused it	`id`, a number	8,802

If you cannot finish the sentence "one row is …" for a table, the table is wrong. `Payment` and `StatusLog` are tables, not columns, because one bill can have several of each (Table 5.4). Book: Section 5.4.4.

The Amount Charged and the Balance

`Bill.payment` is stored although `Service` has the price, because it is *the amount charged on that day* (Equation (5.1)):

$$payment = price \times unit \tag{5.1}$$

The money still owing follows from it and the `Payment` table (Equation (5.2)):

$$balance = payment - \sum_{p \in Payment, p.Bill_id = id} p.amount \tag{5.2}$$

If wash, dry and fold rises from Rp 8.000 to Rp 9.000, last year's receipts must not change: a historical value is recorded, not looked up. Book: Section 5.4.4.

Worked Example 1: the Counter Book

Listing 5.7 rebuilds the one-table counter book from the real data, then makes one ordinary change.

Listing 5.7 `sql/20_flat_table.sql`: the counter book rebuilt from the real data, and one ordinary change to it. `CREATE TABLE ... AS SELECT` makes a new table from the answer to a query.

```sql
-- The "counter book": one wide table in which every line repeats the customer
-- and the service. Built from the real data so that we can watch it go wrong.
CREATE TABLE CounterBook AS
SELECT b.id AS bill, c.name AS customer, c.phone, c.address,
       s.id AS service, s.description, s.price, b.unit, b.payment
  FROM Bill b
  JOIN Customer c ON c.id = b.Customer_id
  JOIN Service  s ON s.id = b.Service_id
 WHERE b.Customer_id = 'C0158' AND b.arrived >= '2026-09-15';

SELECT bill, customer, phone, service, price, unit, payment FROM CounterBook;

-- Update anomaly: the customer changes her phone number, and one line is fixed.
UPDATE CounterBook SET phone = '0813-0000-9999' WHERE bill = 'B2930';
SELECT DISTINCT customer, phone FROM CounterBook;
```

`CREATE TABLE ... AS SELECT` makes a new table from the answer to a query; the customer and the price are repeated on every line. Book: Section 5.4.5.

Worked Example 1: One Change, Two Truths

```
Query OK, 1 row affected

+------------------+----------------+
| customer         | phone          |
+------------------+----------------+
| Endah Ika Rahman | 0877-0000-2053 |
| Endah Ika Rahman | 0813-0000-9999 |
+------------------+----------------+
```

> **THE CURE, NORMALIZATION.** A fact about *one thing* was stored in a row about *another thing*. Put each fact in the table about the thing it describes, and connect the tables with keys.

*After one reasonable `UPDATE`, Endah Ika Rahman has two telephone numbers: the **update anomaly**; deletion and insertion anomalies are its relatives. 1NF: one value per cell; 2NF: depend on the whole key; 3NF: no column depends on another non-key column. Book: Section 5.4.5.*

Types Are Promises, Constraints Are Rules

Table 5.5 The MySQL types used by the laundry. Choosing the narrowest type that can hold every legal value is documentation that the database enforces.

Type	Holds	Used for
`CHAR(5)`	exactly 5 characters	codes of fixed length: `C0056`, `B2931`, `E0004`
`VARCHAR(50)`	up to 50 characters	names, descriptions, e-mail addresses
`INT`	a whole number, about $\pm 2.1 \times 10^9$	`Bill.unit`, the automatic `id` of `Payment`
`DECIMAL(10,2)`	an exact number, 10 digits, 2 after the point	money: `Bill.payment`, `Payment.amount`
`DATE`	a calendar date	`Customer.birthdate`, `Outlet.opened`
`DATETIME`	a date and a time	`Bill.arrived`, `Payment.paid_at`
`ENUM('a','b')`	one of a fixed list of words	`Bill.status`, `Payment.method`

DEFINITION. `NULL` means *no value is recorded*. It equals nothing, not even `NULL`: write `finished IS NULL`, never `finished = NULL`.

Table 5.5: money is `DECIMAL`, never `FLOAT`; a telephone number is text. Constraints (`NOT NULL`, `PRIMARY KEY`, `UNIQUE`, `FOREIGN KEY`, `CHECK`) are rules the database will not let anybody break. Book: Sections 5.5.3 and 5.5.4.

The Heart of the Schema: `Bill`

Listing 5.8 The constraints of the `Bill` table in `sql/24_create_bill.mysql.sql`: the shop's business rules, written once and enforced for every program.

```
    CONSTRAINT Bill_pk PRIMARY KEY (id),
    CONSTRAINT Bill_id_len         CHECK (CHAR_LENGTH(id) = 5),
    CONSTRAINT Bill_status_enum    CHECK (status IN ('departed','finished','queued')),
    CONSTRAINT Bill_unit_pos       CHECK (unit > 0),
    CONSTRAINT Bill_payment_nonneg CHECK (payment >= 0),
    CONSTRAINT Bill_finished_order CHECK (finished IS NULL OR finished >= arrived),
    CONSTRAINT Bill_departed_order CHECK (departed IS NULL OR (departed >= arrived
                                       AND (finished IS NULL OR departed >= finished))),
    CONSTRAINT Bill_status_times   CHECK (
           (status = 'queued'   AND finished IS NULL     AND departed IS NULL)
        OR (status = 'finished' AND finished IS NOT NULL AND departed IS NULL)
        OR (status = 'departed' AND departed IS NOT NULL))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

Above the constraints of Listing 5.8 are twelve columns: `arrived` defaults to now, `finished` and `departed` stay `NULL`. The last three `CHECK`s encode the life of a bill: it cannot finish before it arrived, and its status must match its times. A program that sets `finished` and forgets the time is stopped by the database itself. Book: Section 5.5.5.

Foreign Keys and `ON DELETE`

Listing 5.9 `sql/25_bill_foreign_keys.mysql.sql`: the four foreign keys of `Bill`, in the style of the original design, one `ALTER TABLE` per reference.

```sql
-- Reference: Bill_Customer (original)
ALTER TABLE Bill ADD CONSTRAINT Bill_Customer FOREIGN KEY (Customer_id)
    REFERENCES Customer (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: Bill_Employee (original)
ALTER TABLE Bill ADD CONSTRAINT Bill_Employee FOREIGN KEY (Employee_id)
    REFERENCES Employee (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: Bill_Service (original)
ALTER TABLE Bill ADD CONSTRAINT Bill_Service FOREIGN KEY (Service_id)
    REFERENCES Service (id) ON DELETE RESTRICT ON UPDATE CASCADE;
-- Reference: Bill_Outlet (new)
ALTER TABLE Bill ADD CONSTRAINT Bill_Outlet FOREIGN KEY (Outlet_id)
    REFERENCES Outlet (id) ON DELETE RESTRICT ON UPDATE CASCADE;
```

DEFINITION. `ON DELETE RESTRICT` refuses to delete a row others refer to; `CASCADE` deletes the referring rows too; `SET NULL` forgets the reference. `ON UPDATE CASCADE` copies a changed key into every referring row.

Read Listing 5.9 as sentences: every `Bill.Customer_id` must be an `id` in `Customer`. The laundry chooses `RESTRICT` everywhere: when the database refuses, a human decides. Book: Section 5.5.6.

`INSERT`: a New Customer Walks In

Listing 5.10 `sql/06_insert.sql`: one new customer and one new bill, then a look at both.

```sql
-- A new customer walks in. Name the columns, then give the values in the same order.
INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
                      enrolled, email, password, credit)
VALUES ('C0257', 'Rina Wulandari', 'f', 'Surabaya', '2001-04-17',
        'Jl. Keputih Tegal No. 42, Sukolilo, Surabaya', '0813-0000-4257',
        '2026-09-30 20:15:00', 'rina.wulandari@example.com', '*****', 0);

-- Her first bill: 3 kg of wash, dry & fold at Keputih, received by cashier E0008.
INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                  Customer_id, Employee_id, Service_id, Outlet_id)
VALUES ('B2954', '2026-09-30 20:20:00', 'queued', 'A02', 3, 24000,
        'C0257', 'E0008', 'S0001', 'O02');

SELECT id, name, phone, credit FROM Customer WHERE id = 'C0257';
SELECT id, status, rack, unit, payment, Customer_id FROM Bill WHERE id = 'B2954';
```

Always name the columns in Listing 5.10; omitted columns take their default. The bill's 24000 is Equation (5.1) by hand: $8000 \times 3 = 24000$. *The receipt:* `Query OK, 1 row affected`. *Book: Section 5.7.1.*

When the Rules Are Broken

The chapter's four forbidden statements try a duplicate `C0001`, a bill for `C9999`, a bill for zero kilograms, and deleting a customer who has bills. MariaDB refuses all four:

```
ERROR: (1062, "Duplicate entry 'C0001' for key 'PRIMARY'")

ERROR: (1452, 'Cannot add or update a child row: a foreign key constraint fails
(`profitina_laundry`.`Bill`, CONSTRAINT `Bill_Customer` FOREIGN KEY (`Customer_id`) REFERENCES
`Customer` (`id`) ON UPDATE CASCADE)')

ERROR: (4025, 'CONSTRAINT `Bill_unit_pos` failed for `profitina_laundry`.`Bill`')

ERROR: (1451, 'Cannot delete or update a parent row: a foreign key constraint fails
(`profitina_laundry`.`Bill`, CONSTRAINT `Bill_Customer` FOREIGN KEY (`Customer_id`) REFERENCES
`Customer` (`id`) ON UPDATE CASCADE)')
```

> **THREE ERRORS WORTH MEMORIZING.** 1062 duplicate key; 1452 a foreign key points at nothing; 1451 a delete blocked by rows that point here. PDO reports all three as SQLSTATE `23000`.

4025 names the failed `CHECK`, `Bill_unit_pos`, which is why constraints are worth naming; SQLite says `UNIQUE constraint failed`, `FOREIGN KEY constraint failed` and `CHECK constraint failed`. Each refusal is a bug caught before it became wrong data. Book: Section 5.7.2.

`UPDATE` and `DELETE`, Safely

Listing 5.11 `sql/08_update.sql`: look, change, look again, and try the same change a second time.

```sql
-- Look first: the rows the UPDATE is about to change.
SELECT id, status, finished FROM Bill WHERE id = 'B2909';

-- Bill B2909 has been washed: move it from queued to finished.
UPDATE Bill
   SET status = 'finished', finished = '2026-09-30 20:40:00'
 WHERE id = 'B2909' AND status = 'queued';

-- And look again.
SELECT id, status, finished FROM Bill WHERE id = 'B2909';

-- The same statement a second time changes nothing: the bill is no longer queued.
UPDATE Bill
   SET status = 'finished', finished = '2026-09-30 20:40:00'
 WHERE id = 'B2909' AND status = 'queued';
```

An `UPDATE` without `WHERE` changes every row. Write the `WHERE` first, run it as a `SELECT`, then turn it into an `UPDATE` or a `DELETE`.

*In Listing 5.11 the `WHERE` also names the state, so the second identical `UPDATE` reports **0 rows**: someone else already finished this bill. A bill with payments cannot be deleted (`Payment_Bill` is `RESTRICT`); a cancelled bill keeps its row. Book: Sections 5.7.3 and 5.7.4.*

`SELECT ... WHERE`: the Keputih Queue

Listing 5.12 `sql/03_where.sql`: the queue of the Keputih outlet, oldest first.

```sql
-- Bills still waiting to be washed at the Keputih outlet (O02).
SELECT id, arrived, unit, payment, Customer_id
  FROM Bill
 WHERE status = 'queued' AND Outlet_id = 'O02'
 ORDER BY arrived;
```

Listing 5.12 returns nine rows, `B2909` to `B2951`: nine bags wait on nine racks. `AND`, `OR`, `NOT`, `BETWEEN`, `IN`, `LIKE 'Ayu%'` and `IS NULL` choose rows; equality is one `=`. The clauses are written `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`, but evaluated from `FROM` onward. Book: Sections 5.8.1–5.8.3.

Joins: One Receipt from Three Tables

DEFINITION. A **join** pairs a row of one table with a row of another wherever the `ON` condition, almost always "foreign key equals primary key", is true.

Listing 5.13 `sql/10_join.sql`: one bill with its customer's name and its service's description and price.

```
-- One bill, with the customer's name and the service's description.
SELECT b.id, c.name, s.description, b.unit, s.price, b.payment
  FROM Bill b
  JOIN Customer c ON c.id = b.Customer_id
  JOIN Service  s ON s.id = b.Service_id
 WHERE b.id = 'B2931';
```

```
+-------+-----------------+---------------------------+------+-------+---------+
| id    | name            | description               | unit | price | payment |
+-------+-----------------+---------------------------+------+-------+---------+
| B2931 | Gilang Nasution | Wash, dry & fold (per kg) |    9 |  8000 |   72000 |
+-------+-----------------+---------------------------+------+-------+---------+
1 row in set
```

In Listing 5.13, `b`, `c` and `s` are aliases. Because `Customer.id` is a primary key, each bill finds exactly one customer. $8000 \times 9 = 72000$ *checks Equation (5.1) again. Book: Sections 5.9.1 and 5.9.2.*

`LEFT JOIN`: Keep the Rows with No Partner

Listing 5.14 `sql/12_left_join.sql`: every customer and how many bills they have, fewest first. The query uses `GROUP BY`, explained in Section 5.10.

```sql
-- Every customer with how many bills they have, fewest first.
SELECT c.id, c.name, COUNT(b.id) AS bills
  FROM Customer c
  LEFT JOIN Bill b ON b.Customer_id = c.id
 GROUP BY c.id, c.name
 ORDER BY bills, c.id
 LIMIT 5;
```

```
+-------+-----------------+-------+
| id    | name            | bills |
+-------+-----------------+-------+
| C0020 | Retno Rahman    |     0 |
| C0051 | Felicia Wirawan |     0 |
| C0093 | Rafi Chandra    |     0 |
| C0063 | Teguh Kurniawati|     1 |
| C0102 | Ilham Handoko   |     1 |
+-------+-----------------+-------+
5 rows in set
```

Listing 5.14 finds three customers who never brought a bag, which an inner join cannot. Count `COUNT(b.id)`, not `COUNT(*)`: the `NULL` row of a customer without bills would count as 1. Book: Section 5.9.3.

`GROUP BY`: What Earns the Money?

> **DEFINITION.** An **aggregate** (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) turns many rows into one value. `GROUP BY c` makes one pile per value of `c` and aggregates each pile.

Listing 5.15 `sql/14_group_service.sql`: revenue per service for the whole year.

```sql
-- Revenue per service, most valuable first.
SELECT s.id, s.description,
       COUNT(*)       AS bills,
       SUM(b.unit)    AS units,
       SUM(b.payment) AS revenue
  FROM Bill b
  JOIN Service s ON s.id = b.Service_id
 GROUP BY s.id, s.description
 ORDER BY revenue DESC;
```

Every column in the `SELECT` list of Listing 5.15 is grouped by or aggregated. Wash, dry and fold earns Rp 67.632.000, half the business; check ironing by hand: $1251 \times 6000 = 7{,}506{,}000$. *Book: Sections 5.10.1 and 5.10.2.*

Worked Example 2: Who Still Owes Money?

Listing 5.16 `sql/17_balance.sql`: bills with money owing, largest balance first. The sums of payments are computed per bill in a subquery before they are joined.

```sql
-- Bills that still have money owing: charged minus everything paid so far.
SELECT b.id, c.name, b.payment AS charged,
       COALESCE(p.paid, 0) AS paid,
       b.payment - COALESCE(p.paid, 0) AS balance
  FROM Bill b
  JOIN Customer c ON c.id = b.Customer_id
  LEFT JOIN (SELECT Bill_id, SUM(amount) AS paid
               FROM Payment
              GROUP BY Bill_id) p ON p.Bill_id = b.id
 WHERE b.payment - COALESCE(p.paid, 0) > 0
 ORDER BY balance DESC, b.id
 LIMIT 6;
```

Listing 5.16 is Equation (5.2) in SQL. The subquery sums payments per bill before the join: joining `Payment` and `StatusLog` directly would multiply rows, $2 \times 3 = 6$. `LEFT JOIN` keeps unpaid bills; `COALESCE` turns their `NULL` into 0. Book: Section 5.10.5.

Worked Example 2: the Answer

```
+-------+--------------------+---------+------+---------+
| id    | name               | charged | paid | balance |
+-------+--------------------+---------+------+---------+
| B1542 | Farah Putri Kusuma |   90000 |    0 |   90000 |
| B0368 | Umi Permana        |   72000 |    0 |   72000 |
| B1021 | Laras Irawan       |   72000 |    0 |   72000 |
| B2931 | Gilang Nasution    |   72000 |    0 |   72000 |
| B1493 | Nadia Umi Susanto  |   64000 |    0 |   64000 |
| B2941 | Farah Wijaya       |   64000 |    0 |   64000 |
+-------+--------------------+---------+------+---------+
6 rows in set
```

Check `B2931` by hand: $9 \times 8000 = 72000$ charged, nothing paid, so the balance is $72000 - 0 = 72000$.

In the whole year 38 bills have a balance, Rp 1.529.000 in total, most of them still in the shop. Book: Section 5.10.5.

Transactions

DEFINITION. A **transaction** is a group of statements treated as one indivisible act: `BEGIN` opens it, `COMMIT` makes every change permanent at once, `ROLLBACK` undoes every change. **ACID:** atomic, consistent, isolated, durable.

Listing 5.17 The transaction of `sql/18_transaction_commit.sql`: a bill, its first log entry and a down payment, together or not at all.

```sql
-- A new bill is three rows: the bill, its first status-log entry, and a down payment.
-- They go in together or not at all.
BEGIN;
INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                  Customer_id, Employee_id, Service_id, Outlet_id)
VALUES ('B2954', '2026-09-30 20:45:00', 'queued', 'A02', 2, 70000,
        'C0094', 'E0004', 'S0008', 'O01');
INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
VALUES ('B2954', 'queued', '2026-09-30 20:45:00', 'E0004');
INSERT INTO Payment (Bill_id, paid_at, method, amount, reference)
VALUES ('B2954', '2026-09-30 20:45:30', 'qris', 35000, 'QRIS 100200300400');
COMMIT;
```

In Listing 5.17 two bed covers cost Rp 70.000 by Equation (5.1); Rp 35.000 is paid; Equation (5.2) leaves Rp 35.000. After `ROLLBACK` a count goes 2,953 before, 2,954 inside, 2,953 after. Book: Sections 5.11.1 and 5.11.2.

PHP and MySQL the Classic Way

DEFINITION. A **result set** is the answer to a `SELECT`, read one row at a time through a cursor; each fetch returns the row and moves on, and `null` when none is left.

Listing 5.18 From `php/mysqli/query.php`: `mysqli_query`, a check for failure, and a `while` loop over `fetch_row()`.

```
$query  = "SELECT description, price FROM Service WHERE price >= 40000 ORDER BY price, id";
$result = mysqli_query($conn, $query);
if (!$result) {
    die("Database access failed " . mysqli_error($conn) . PHP_EOL);
} else {
    echo "Executing query success!", PHP_EOL;
}

echo PHP_EOL, "--- fetch_row(): a row is a numbered array ---", PHP_EOL;
while ($row = $result->fetch_row()) {
    echo "$row[0] costs Rp $row[1]", PHP_EOL;
}
```

Listing 5.18 reads four rows as `$row[0]`; after `data_seek(0)` the program reads them again as `$row['description']` with `MYSQLI_ASSOC`; prefer names. `free()` and `mysqli_close()` clean up. Settings come from `config.php`, never typed into the program. Book: Sections 5.12.1–5.12.4.

PDO: One Interface, Many Databases

DEFINITION. PDO (PHP Data Objects) is one interface to MySQL, MariaDB, SQLite, PostgreSQL and others; only the **DSN**, the data source name, changes.

Listing 5.19 The start of `connect()` in `php/pdo/Db.php`: three options that fix bad defaults, and the MySQL DSN.

```
function connect(bool $fresh = false): PDO
{
    $cfg = require __DIR__ . '/../config.php';
    $options = [
        PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,   // failures throw
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,         // rows keyed by column name
        PDO::ATTR_EMULATE_PREPARES   => false,                    // real prepared statements
    ];

    if ($cfg['engine'] === 'mysql') {
        $dsn = sprintf('mysql:host=%s;port=%d;dbname=%s;charset=utf8mb4',
                       $cfg['host'], $cfg['port'], $cfg['database']);
        $pdo = new PDO($dsn, $cfg['user'], $cfg['password'], $options);
        $pdo->exec("SET SESSION sql_mode = 'STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION'");
        return $pdo;
    }
```

Listing 5.19: failures throw, rows are keyed by name, prepared statements are real. The SQLite branch copies `laundry.db` and runs `PRAGMA foreign_keys = ON`. Pattern: `prepare` with `?`, `execute([...])`, then `fetch`, `fetchAll` or `rowCount`. With the silent error mode a failure is a `false` nobody checks. Book: Sections 5.14.1, 5.14.2 and 5.14.5.

Customers: Create and Read

Listing 5.20 `createCustomer()` from `php/pdo/Customers.php`: nine values bound to nine placeholders.

```
function createCustomer(PDO $pdo, string $name, string $gender, string $place,
                        string $birthdate, string $address, string $phone,
                        string $email, string $now): string
{
    $id = nextCustomerId($pdo);
    $stmt = $pdo->prepare(
        "INSERT INTO Customer (id, name, gender, place, birthdate, address, phone,
                               enrolled, email, password, credit)
         VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, '*****', 0)");
    $stmt->execute([$id, $name, $gender, $place, $birthdate, $address, $phone, $now, $email]);
    return $id;
}
```

In Listing 5.20, `'*****'` and `0` are the programmer's constants, not input; `findCustomer()`, `updateCustomerPhone()` and `deleteCustomer()` have the same shape. `nextCustomerId()` turns `C0256` into `C0257`. Search binds `'%' . $text . '%'` and `LIMIT` as `PDO::PARAM_INT`; deleting `C0094`, who has bills, is refused with SQLSTATE `23000`. Book: Section 5.14.3.

`createBill()`: the Price Is Computed

Listing 5.21 The start of `createBill()` in `php/pdo/Bills.php`: begin, read the price, apply the price formula.

```
function createBill(PDO $pdo, string $customerId, string $serviceId, int $unit,
                    string $rack, string $outletId, string $employeeId, string $now,
                    ?string $payMethod = null, int $payAmount = 0): string
{
    $pdo->beginTransaction();
    try {
        $price = $pdo->prepare("SELECT price FROM Service WHERE id = ?");
        $price->execute([$serviceId]);
        $unitPrice = $price->fetchColumn();
        if ($unitPrice === false) {
            throw new InvalidArgumentException("no such service: $serviceId");
        }
        $charged = (int) $unitPrice * $unit;        // the price is computed here, never typed in
```

The price is never an argument of Listing 5.21: `createBill()` reads it from `Service` and applies Equation (5.1) itself, so no form can send a bill for Rp 1. Next comes the `INSERT` of the `Bill` row itself. Book: Section 5.14.4.

`createBill()`: Commit or Roll Back

Listing 5.22 The end of `createBill()`: the status log, the optional payment, `commit()`, and `rollBack()` on any `Throwable`.

```
        $pdo->prepare(
            "INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
             VALUES (?, 'queued', ?, ?)")
            ->execute([$id, $now, $employeeId]);
        if ($payMethod !== null) {
            $pdo->prepare(
                "INSERT INTO Payment (Bill_id, paid_at, method, amount) VALUES (?, ?, ?, ?)")
                ->execute([$id, $now, $payMethod, $payAmount]);
        }
        $pdo->commit();                                  // all three tables, at once
        return $id;
    } catch (Throwable $e) {
        $pdo->rollBack();                                // none of them, as if nothing happened
        throw $e;
    }
}
```

The shape of Listing 5.22 never changes: `beginTransaction()`; the work inside `try`; `commit()` last; `rollBack()` and `throw $e` in `catch (Throwable $e)`. Book: Section 5.14.4.

The Life of a Bill, Run

```
before: Bill=2953 StatusLog=8802 Payment=3319

1. CREATE a bill: 2 bed covers, half paid now by QRIS
  B2954  queued    Khadijah Hakim, 2 x Bed cover, any size (per item)
           charged Rp 70.000, paid Rp 35.000
2. UPDATE: the washer finishes it
bool(true)
   ...and a second time (it is no longer queued)
bool(false)
3. The customer pays the rest in cash
   payment id 3321
  B2954  finished  Khadijah Hakim, 2 x Bed cover, any size (per item)
           charged Rp 70.000, paid Rp 70.000
4. A bill whose payment method is not allowed (gopay)
   refused: SQLSTATE[23000]: Integrity constraint violation: 19 CHECK constraint failed:
Payment_method_enum
   everything rolled back

after : Bill=2954 StatusLog=8804 Payment=3321
```

Step 4 is the one to study: `Bill` and `StatusLog` were written, the payment failed because `gopay` is not allowed, and the counts show nothing of that bill remains. `finishBill()` returns `false` the second time. Book: Section 5.14.4.

A Sign-In Page That Is Wide Open

Listing 5.23 From `php/injection/unsafe.php`: a sign-in check and a bill-status lookup that glue what the visitor typed into the SQL text.

```
// DO NOT COPY THIS FILE INTO A REAL APPLICATION. It glues what a stranger typed
// into the SQL text, which is exactly the mistake this section demonstrates.

function loginUnsafe(PDO $pdo, string $email, string $password, ?string &$sql = null): ?array
{
    $sql = "SELECT id, name, email FROM Employee
              WHERE email = '$email' AND password = '$password'";
    $row = $pdo->query($sql)->fetch();
    return $row === false ? null : $row;
}

function billStatusUnsafe(PDO $pdo, string $billId, ?string &$sql = null): array
{
    $sql = "SELECT id, status FROM Bill WHERE id = '$billId'";
    return $pdo->query($sql)->fetchAll(PDO::FETCH_NUM);
```

*Nothing in Listing 5.23 looks alarming. A query has **instructions**, which you wrote, and **data**, which a stranger typed; gluing strings mixes them, and the database cannot tell which part came from whom. Book: Section 5.15.1.*

Eleven Characters

```
=== password box attack
typed e-mail   : kasir1@profitina-laundry.example
typed password : ' OR '1'='1
SQL built      : SELECT id, name, email FROM Employee WHERE email = 'kasir1@profitina-laundry.example'
AND password = '' OR '1'='1'
unsafe page    : LET IN as Hendra Wijaya (E0001)
safe page      : REFUSED

=== e-mail box attack
typed e-mail   : kasir1@profitina-laundry.example' --
typed password : anything
SQL built      : SELECT id, name, email FROM Employee WHERE email = 'kasir1@profitina-laundry.example'
-- ' AND password = 'anything'
unsafe page    : LET IN as Dewi Lestari Putri (E0004)
safe page      : REFUSED
```

The first apostrophe closes the string; the rest continues as SQL (Equation (5.3)):

$$(email = \text{'kasir1@...'} \wedge password = \text{''}) \vee (\text{'1'} = \text{'1'}) \tag{5.3}$$

`AND` binds before `OR`: the left half is false for every row, the right half true for every row, so the first employee, the owner, is let in. `--` plus a space comments out the password test. Book: Sections 5.15.1 and 5.15.2.

Reading Tables the Page Never Mentions

```
=== UNION attack
typed bill id  : x' UNION SELECT name, phone FROM Customer ORDER BY 1 LIMIT 3 --
SQL built      : SELECT id, status FROM Bill WHERE id = 'x' UNION SELECT name, phone FROM Customer ORDER
BY 1 LIMIT 3 -- '
unsafe page    : 3 row(s)
                 Adi Eko Alatas | 0811-0000-1612
                 Adi Saputra | 0811-0000-1509
                 Agus Fauzi | 0856-0000-8730
safe page      : 0 row(s)
```

> **This is not a clever trick, and it is not rare.** Injection has been on the OWASP Top Ten since 2003 and is still there in 2025, as A05; every attack here was typed into an ordinary form field.

`UNION` appends the rows of the stranger's own `SELECT`: customers' names and telephone numbers, which are personal data under UU PDP No. 27/2022. Book: Section 5.15.2.

The Repair: Prepared Statements

DEFINITION. A **prepared statement** has a placeholder (`?` or `:name`) wherever a value goes, and is sent to the database *first*. The values follow separately, as data, so no value can change what the statement does.

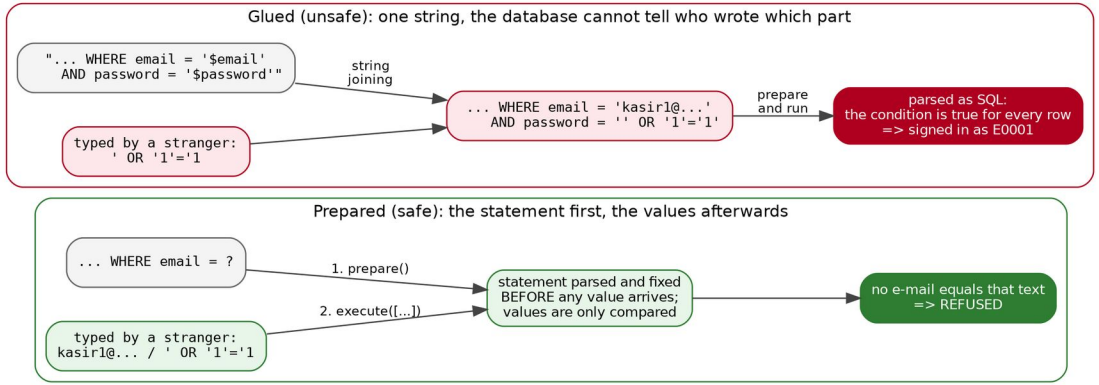


Figure 5.5 Glued SQL against a prepared statement. When strings are glued, the stranger's text becomes part of the statement; when the statement is prepared first, the same text can only ever be compared, as a value.

Figure 5.5: when strings are glued the stranger's text becomes part of the statement; when the statement is prepared first, the same text can only ever be compared, as a value. Book: Section 5.15.3.

The Repaired Functions

Listing 5.24 From `php/injection/safe.php`: find the employee by e-mail alone, check the password against the stored hash, and look up a bill with a placeholder.

```
function loginSafe(PDO $pdo, string $email, string $password): ?array
{
    $stmt = $pdo->prepare("SELECT id, name, email, password_hash FROM Employee WHERE email = ?");
    $stmt->execute([$email]);
    $row = $stmt->fetch();
    if ($row === false || $row['password_hash'] === null
            || !verifyPassword($password, $row['password_hash'])) {
        return null;
    }
    unset($row['password_hash']);                    // never hand the hash onward
    return $row;
}

function billStatusSafe(PDO $pdo, string $billId): array
{
    $stmt = $pdo->prepare("SELECT id, status FROM Bill WHERE id = ?");
    $stmt->execute([$billId]);
    return $stmt->fetchAll(PDO::FETCH_NUM);
}
```

*With Listing 5.24 the honest password works and all three attacks are refused; the `UNION` finds **0 rows**, because the database searched for a bill whose id is the literal attack text. `verifyPassword()` uses PBKDF2 and the constant-time `hash_equals()`. Book: Section 5.15.3.*

The Rule

> **THE RULE.** Every value that came from outside your program goes into SQL as a bound parameter. Every single one, not only the ones that look dangerous.

A table or column name cannot be a placeholder: compare the visitor's choice with a list you wrote, and insert only your own string.

`LIKE` wildcards: binding `'%' . $text . '%'` is safe; a `%` in the visitor's text only broadens their own search.

`mysqli` has prepared statements too: `$stmt->bind_param('s', $email)`, one type letter per placeholder.

Escaping with `mysqli_real_escape_string()` must be remembered for every value, forever, by everyone; a prepared statement is right by construction. Book: Section 5.15.4.

A Small Web Application: Sign-In

Listing 5.25 The PHP half of `php/web/login.php`: `loginSafe()`, a fresh session id at sign-in, and a redirect.

```php
<?php
// login.php: sign in with an e-mail and a password; the identity goes into the session.
declare(strict_types=1);
require __DIR__ . '/boot.php';
require __DIR__ . '/../injection/safe.php';

$error = '';
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $user = loginSafe(connect(), (string) ($_POST['email'] ?? ''), (string) ($_POST['password'] ?? ''));
    if ($user !== null) {
        session_regenerate_id(true);          // a new ticket at sign-in: an old one is worthless
        $_SESSION['user'] = $user;
        header('Location: customers.php');
        exit;
    }
    $error = 'Wrong e-mail or password.';
}
```

*In Listing 5.25, `session_regenerate_id(true)` makes any id planted before sign-in worthless. The protected page answers a successful `POST` with a redirect (Post/Redirect/Get), gives an invalid form **400**, and escapes everything with `h()`. Book: Section 5.16.*

The Same Transaction in Java: JDBC

Listing 5.26 The end of `finish` in `java/BillDao.java`: `commit()`, `rollback()`, and auto-commit restored in `finally`.

```
            if (changed != 1) {                                  // not queued: nothing to finish
                conn.rollback();
                return false;
            }
            log(id, "finished", now, employeeId);
            conn.commit();
            return true;
        } catch (SQLException e) {
            conn.rollback();
            throw e;
        } finally {
            conn.setAutoCommit(true);
        }
```

JDBC: *`Connection`, `PreparedStatement`, `ResultSet`; placeholders are `?`, numbered from **1**, bound with `setString`, `setInt`, `setLong`. The method opens with `conn.setAutoCommit(false)`, which is `BEGIN`; Listing 5.26 ends the transaction either way. Book: Sections 5.18.1 and 5.18.2.*

The Same Transaction in Python

Listing 5.27 The body of `create_bill` in `python/bills.py`: the DB-API opens the transaction by itself; the program only ends it.

```
    try:
        price = run(conn, "SELECT price FROM Service WHERE id = ?", (service_id,)).fetchone()
        if price is None:
            raise ValueError("no such service: " + service_id)
        bid = next_bill_id(conn)
        run(conn, """INSERT INTO Bill (id, arrived, status, rack, unit, payment,
                                       Customer_id, Employee_id, Service_id, Outlet_id)
                     VALUES (?, ?, 'queued', ?, ?, ?, ?, ?, ?, ?)""",
            (bid, now, rack, unit, int(price["price"]) * unit,
             customer_id, employee_id, service_id, outlet_id))
        run(conn, """INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id)
                     VALUES (?, 'queued', ?, ?)""", (bid, now, employee_id))
        if pay_method is not None:
            add_payment(conn, bid, pay_method, pay_amount, now, commit=False)
        conn.commit()                                  # all three tables, at once
        return bid
    except Exception:
        conn.rollback()                                # none of them
        raise
```

Listing 5.27 is `createBill()` line for line; `run()` in `db.py` turns `?` into `%s` for PyMySQL, a placeholder, not Python's `%` operator. Flask's `request.args` is `$_GET`, `request.form` is `$_POST`, and Jinja's `{{ }}` escapes automatically. Book: Section 5.19.

One Story, Six Runs

Table 5.6 Six real runs of the same CRUD story, in three languages on two database engines. The rules live in the schema and in one small data layer per language; everything else asks the database.

Language	Driver	Database	New customer	Bill charged / paid	Attacks refused
PHP 8.4	PDO	SQLite 3.45	`C0257`	Rp 70.000 / Rp 70.000	all
PHP 8.4	PDO, `mysqli`	MariaDB 10.11	`C0257`	Rp 70.000 / Rp 70.000	all
Java 21	sqlite-jdbc 3.53.2.0	SQLite	`C0257`	Rp 70.000 / Rp 70.000	all
Java 21	MariaDB Connector/J 2.7.6	MariaDB 10.11	`C0257`	Rp 70.000 / Rp 70.000	all
Python 3.11	`sqlite3`	SQLite 3.45	`C0257`	Rp 70.000 / Rp 70.000	all
Python 3.11	PyMySQL 2.2	MariaDB 10.11	`C0257`	Rp 70.000 / Rp 70.000	all

Table 5.6: PHP, Java and Python, on SQLite and on MariaDB, print the same new customer, the same Rp 70.000 bill, and refuse the same attacks. The rules live in the schema. Book: Section 5.19.3.

Profitina in Practice

Profitina (`app.profitina.com`, Surabaya) measures how ready a brand's website is to be cited by AI answer engines: Python on **PostgreSQL**, SQLite in parts of the test path.

Every report is a `GROUP BY` over a growing table: the weekly visibility trend is the monthly report with other columns.

Prepared statements, without exception: addresses and posts typed by anyone would otherwise hand over the database.

Personal data is law: consent, export and complete account deletion under UU PDP No. 27/2022; deletion is the `ON DELETE` question asked by a regulator.

Runnable practice always uses Profitina Laundry; what carries over is the shape: a normalized schema, bound parameters, transactions, an index on the column you search by. Book: Section 5.20.

Common Mistakes

Configuration: editing `httpd.conf` or `php.ini` without restarting, or the wrong `php.ini`; "port 80 in use"; double-clicking a `.php` file.

State: output before `header()`; no `exit` after a redirect; expecting a new cookie in `$_COOKIE`; trusting a cookie with an identity.

SQL: `bill` for `Bill`; `= NULL`; `UPDATE` without `WHERE`; `LIMIT` without `ORDER BY`; `COUNT(*)` after a `LEFT JOIN`; money in `FLOAT`.

Code: gluing values into SQL (also with Python's `%` and f-strings); a named placeholder used twice; several writes without a transaction; connecting as `root`.

Each mistake is listed in the book with its fix. Book: Section 5.21.

Summary: Five Things to Remember

1. A web server is three programs, Apache, PHP and MySQL, each with its own configuration file; the document root maps a folder onto a website.

2. HTTP has no memory: a cookie keeps data on the client, a session keeps it on the server and sends only an id. Identity belongs in a session.

3. Tables connect only by matching values; keys and constraints keep the data true; Equation (5.1) and Equation (5.2) are stored and derived facts.

4. `SELECT ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT`, with joins; several writes that belong together go in one transaction.

5. Every outside value is a bound parameter, in PHP, Java and Python alike.

One idea, three languages, the same numbers. Book: Section 5.22.

Next Time

Chapter 6 — XML and Structured Data. Well-formed and valid documents, DTD and XML Schema, DOM, SAX and XPath, XSLT, and JSON.

The bills we query today leave the database as an XML and a JSON feed, built in PHP, Java and Python.

Before then: work Exercises 5.3, 5.7, 5.9, 5.12 and 5.15.

Solutions to every exercise are in the book, with the real output of each query. Book: Sections 5.23 and 5.24.

Lecture 6 — XML and Structured Data

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 6

Lecturer: Irfan Subakti — yifana@gmail.com

Today the human leaves the page: a program writes the data and a different program reads it. Book: Chapter 6, Sections 6.1–6.24.

What This Meeting Covers

1. **Why structured data** — two programs that must agree; names that travel with values.
2. **XML itself** — a meta-language, well-formedness, entities, elements or attributes, the tree, namespaces.
3. **Grammars** — validity, the DTD, XML Schema, validating in PHP, Java and Python.
4. **Using documents** — DOM, SAX and pull parsing; XPath; XSLT; writing XML with a library.
5. **JSON and the feed** — six types, what was gained and lost, a feed endpoint, and the entity traps.

Every example uses one real day: outlet O01 (Rungkut) on 26 September 2026, four bills. Book: Section 6.1.

Where We Left Off

Chapter 5 gave Profitina Laundry a real database, `laundry.db`, with seven tables: `Outlet`, `Customer`, `Employee`, `Service`, `Bill`, `Payment`, `StatusLog`.

Every chapter so far had **a human being at one end**: somebody clicks, somebody reads. `<h1>` means *draw this large*.

Now the head office wants each outlet's bills every day, an accounting firm once a month, the JavaScript front end the price list. **Nobody looks at the data.**

The questions: in what form does the data leave the database, how do writer and reader agree on it, and how does the reader check what arrived?

This chapter removes the human. Book: Section 6.1.

The Plan of the Chapter

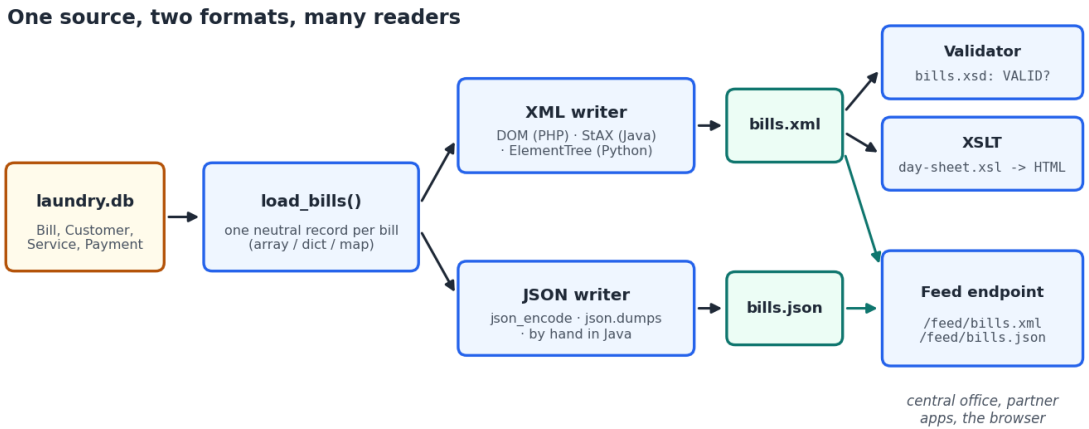


Figure 6.1 The plan of this chapter: one source, two formats, many readers. Every box on the right is built, in PHP, Java and Python, before the chapter ends.

***Figure 6.1**: one query produces one neutral record per bill; two writers produce XML and JSON; the documents are validated, transformed into a day sheet, and served by a feed endpoint. Book: Section 6.1.*

Two Programs That Have to Agree

> **DEFINITION.** A **data interchange format** is an agreed way of writing information down as bytes, so that a program which did not produce it can read it back without asking the producer any questions.

Listing 6.1 The four bills of 26 September 2026 as comma-separated values. Everything a reader needs to know is missing from the file itself.

```
B2905,C0022,Vania Wirawan,S0003,7,departed,42000,cash,42000
B2908,C0121,Laras Chandra,S0001,10,departed,80000,qris,80000
B2911,C0246,Dimas Wibowo,S0009,1,departed,0,,
B2913,C0216,Diah Siregar,S0012,1,finished,30000,credit,30000
```

*Before **Listing 6.1** can be used, a colleague must be told: is column 5 kilograms or items? Why does `42000` appear twice? Why are two columns empty? And Wash, dry & fold would shift every column after its comma. The data and its meaning have come apart. Book: Sections 6.2.1 and 6.2.2.*

Names Travel with Values

Listing 6.2 One bill as XML, shortened. Every value is labeled by the element or attribute that contains it.

```
<bill id="B2911" status="departed" cancelled="true">
  <customer id="C0246">Dimas Wibowo</customer>
  <service id="S0009" price="25000">Blanket (per item)</service>
  <unit>1</unit>
  <charged>0</charged>
</bill>
```

> **KEY INSIGHT.** In CSV the name of a column lives outside the file. In XML the name of every value is written next to it, every time. The cost is repetition; the benefit is a file that still means something in ten years.

*In **Listing 6.2** the `1` is a `unit`, the bill is `cancelled`, which explains the zero, and there is no `payment` because nothing was paid. Java made the application portable; XML makes the data portable. Book: Sections 6.2.2 and 6.2.3.*

A Language for Making Languages

> **DEFINITION.** XML, the **eXtensible Markup Language**, is a **meta-language**: a set of rules for inventing markup languages. It has no tags of its own.

Listing 6.3 HTML: every tag is an instruction about appearance. Make this big, make this bold, start a paragraph.

```
<H1>This is a heading</H1>
<P>A<B>paragraph</B>started from here</P>
```

Listing 6.4 The same content marked up by meaning. Nothing here says “big” or “bold”.

```
<article>
  <title>This is a heading</title>
  <body>A <emphasis>paragraph</emphasis> started from here</body>
</article>
```

Listing 6.3 *says bold;* ***Listing 6.4*** *says this matters. XML describes* ***structure*** *and* ***meaning****, never format; one* ***parser*** *reads MathML, SVG and the Laundry feed alike. Book: Sections 6.3.1 and 6.3.2.*

Five Things XML Is Not

Table 6.1 Five claims about XML, each corrected. XML is not any of these things, but it can be made into, used from, sent over or stored in all of them.

Claim	Why it is wrong	What is true
XML replaces HTML	XML defines no tags, so it cannot draw a page	HTML *can be written as* XML: that is XHTML
XML is a presentation format	XML says nothing about appearance	XML *can be turned into* a presentation format, by XSLT
XML is a programming language	A document has no instructions, variables or loops	XML *can be used from* almost any programming language
XML is a network protocol	It defines no connections, ports or handshakes	XML *can be sent over* a network: SOAP does exactly that
XML is a database	It has no index, no query optimizer, no transaction	XML *can be stored in* a database, and databases can return it

*Every correction in **Table 6.1** turns a claim of identity into a claim of relationship. Open `bills.xml` in a browser and you get a tree of tags: nobody told the browser what a `<bill>` is. Book: Sections 6.3.3 and 6.3.4.*

A First Document

```xml
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<note>
    <to>Yarik</to>
    <from>Naya</from>
    <heading>Reminder</heading>
    <body>Don't forget to help mommy!</body>
</note>
```

Figure 6.2 A complete XML document: a short note from Naya to Yarik. Every rule of the language is visible in these six lines.

*Read **Figure 6.2** aloud: this is a note; its recipient is Yarik; its sender is Naya. If the sentence works, the names are good; `value1`, `value2` would describe nothing. Book: Section 6.4.*

Declaration, Elements, Attributes

Listing 6.5 The XML declaration of **Figure 6.2**. It begins `<?` and ends `?>`, which marks it as a processing instruction rather than an element.

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
```

DEFINITION. An **element** is a start tag, its content and its matching end tag. An **attribute** is a `name="value"` pair inside a start tag. The **root element** contains all the others; every document has exactly one.

Use UTF-8. A file that claims Latin-1 but holds a character outside it will not parse; a file that claims UTF-8 but was saved as Latin-1 may parse and give you the wrong letters.

Listing 6.5 *is a processing instruction, not an element. Book: Sections 6.4.1 and 6.4.2.*

The Bill Feed, Part 1

Listing 6.6 The start of `data/bills.xml`, exported from `laundry.db`: the declaration, a comment, the root with its outlet and date, and bill B2905.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Profitina Laundry: the bills one outlet received on one day.
     Money is whole rupiah. Times are shop time, WIB (UTC+7). -->
<bills outlet="O01" date="2026-09-26">
  <bill id="B2905" status="departed">
    <customer id="C0022">Vania Wirawan</customer>
    <service id="S0003" price="6000">Ironing only (per kg)</service>
    <unit>7</unit>
    <rack>B03</rack>
    <arrived>2026-09-26T09:05:04+07:00</arrived>
    <finished>2026-09-27T13:27:17+07:00</finished>
    <departed>2026-09-27T14:43:55+07:00</departed>
    <charged>42000</charged>
    <payment method="cash" at="2026-09-27T14:43:55+07:00">42000</payment>
  </bill>
```

*Read **Listing 6.6** aloud: bill B2905, departed; customer C0022, Vania Wirawan; ironing only at 6,000 rupiah per kilogram, unit 7; charged 42,000, and 42,000 paid in cash. Book: Section 6.4.3.*

The Bill Feed, Part 2

Listing 6.7 The end of `data/bills.xml`: the cancelled bill B2911 and bill B2913, still waiting to be collected.

```
  <bill id="B2911" status="departed" cancelled="true">
    <customer id="C0246">Dimas Wibowo</customer>
    <service id="S0009" price="25000">Blanket (per item)</service>
    <unit>1</unit>
    <rack>A17</rack>
    <arrived>2026-09-26T17:06:02+07:00</arrived>
    <departed>2026-09-27T07:46:02+07:00</departed>
    <charged>0</charged>
  </bill>
  <bill id="B2913" status="finished">
    <customer id="C0216">Diah Siregar</customer>
    <service id="S0012" price="30000">Helmet cleaning (per item)</service>
    <unit>1</unit>
    <rack>A06</rack>
    <arrived>2026-09-26T18:42:06+07:00</arrived>
    <finished>2026-09-28T16:02:33+07:00</finished>
    <charged>30000</charged>
    <payment method="credit" at="2026-09-26T18:45:55+07:00" reference="DEPOSIT">30000</payment>
  </bill>
</bills>
```

*In **Listing 6.7** B2911 has no `finished` and no `payment`; B2913 has no `departed`. Book: Section 6.4.3.*

Why the Feed Looks Like This

Identifiers are attributes; content is elements. `id`, `status`, `method` are small atomic facts; names, descriptions and amounts are content.

`charged`, not `payment`. The database column `Bill.payment` is the amount charged; each `payment` element is a sum actually received, and may repeat.

Absent means absent. No `finished` element, never an empty `<finished></finished>`; `cancelled` is written only when it is true.

Times carry their zone: `2026-09-26T09:05:04+07:00`, the `xs:dateTime` form. **Only what the reader needs:** no phone, e-mail, birth date or password hash.

A feed is a published promise, so it uses the clearest name for everything. Book: Section 6.4.3.

Worked Example 1: the Day's Arithmetic

EXAMPLE. Every program in the chapter prints these numbers, so check them by hand first.

Charged: $42000 + 80000 + 0 + 30000 = 152000$ rupiah.

Paid: $42000 + 80000 + 30000 = 152000$ rupiah, so nothing is outstanding.

B2908: unit 10 at 8,000 per kilogram, $10 \times 8000 = 80000$, exactly its `charged`: the price was copied when the bill was created.

The test: four bills, one cancelled, Rp 152.000 charged, Rp 152.000 paid. A program that prints that is wired up correctly.

A later change of price never rewrites history. Book: Section 6.4.3.

Well-Formed: Six Rules

> **DEFINITION.** A document is **well-formed** when it obeys XML's syntax. A parser refuses anything else: not warn, not guess, *refuse*.

1. One root element. **2.** Every start tag has an end tag (or is written `<br/>`). **3.** The same capitalization in both.

4. Elements nest, never overlap: the element opened last is closed first. **5.** Attribute values in quotation marks. **6.** No attribute repeated on one element.

HTML forgives `<p>one<p>two`, which is why HTML parsers are enormous; XML forgives nothing, which is why an XML parser is small. Book: Sections 6.5.1–6.5.6.

A Document Broken Four Ways

Listing 6.8 `data/bills-malformed.xml`, broken four ways: an unquoted attribute value (line 2), a case mismatch (line 4), a bare `&` (line 5) and a `unit` element never closed (line 6).

```
<?xml version="1.0" encoding="UTF-8"?>
<bills outlet="O01" date=2026-09-26>
  <bill id="B2908" status="departed">
    <customer id="C0121">Laras Chandra</Customer>
    <service id="S0001" price="8000">Wash, dry & fold (per kg)</service>
    <unit>10
    <charged>80000</charged>
  </bill>
</bills>
```

*Find the four faults in **Listing 6.8** before turning the page: an unquoted `date`, `</Customer>` closing `customer`, a bare `&`, and `<unit>` never closed. Book: Section 6.5.7.*

The Parser Stops at the First Fault

Listing 6.9 What `php/01_wellformed.php` prints for the two files: four messages, all about line 2.

```
bills.xml            WELL-FORMED
bills-malformed.xml  NOT WELL-FORMED
    line 2: AttValue: " or ' expected
    line 2: attributes construct error
    line 2: Couldn't find end of Start Tag bills line 2
    line 2: Extra content at the end of the document
```

The first message is the real one: `AttValue: " or ' expected`. The rest is the wreckage of that single failure. **Fix one thing and run again.**

Listing 6.9 *never mentions lines 4 to 6. Python says* `not well-formed (invalid token)`*; Java says* `Open quote is expected`*. Three libraries, three wordings, one fault. Book: Section 6.5.7.*

Characters That Already Have a Job

DEFINITION. An **entity reference** stands for a piece of text: `<`. XML predefines five. A **character reference** names a code point: `<` or `<`.

Table 6.2 The five entity references predefined by XML, with their character references.

Write this	For	Character	Also legal
`<`	less-than	`<`	`<` or `<`
`>`	greater-than	`>`	`>` or `>`
`&`	ampersand	`&`	`&` or `&`
`"`	quotation mark	`"`	`"` or `"`
`'`	apostrophe	`'`	`'` or `'`

*From **Table 6.2**, `&` is always required, so Wash, dry & fold is written `Wash, dry & fold`; Ma'ruf needs no escaping as content. CDATA and comments are for the typist and the human reader, not for data. Book: Section 6.6.*

Elements or Attributes?

```
<population animal="lion">80</population>

<population>
    <animal>lion</animal>
    <quantity>80</quantity>
</population>
```

Figure 6.3 The same fact, a population of 80 lions, written twice: first with `animal` as an attribute of `population`, then with `animal` as a child element beside a `quantity` element. The two documents carry the same information.

***Figure 6.3**: the same fact two ways. An element can repeat, have parts, grow later, and keeps its order. Use an attribute for a small atomic fact about the thing; when torn, choose the element. Book: Section 6.7.*

Every Document Is a Tree

DEFINITION. If *B* is written directly inside *A*, *A* is the **parent** of *B* and *B* a **child** of *A*. Elements with the same parent are **siblings**; below *A* at any depth are its **descendants**. Every element except the root has exactly one parent.

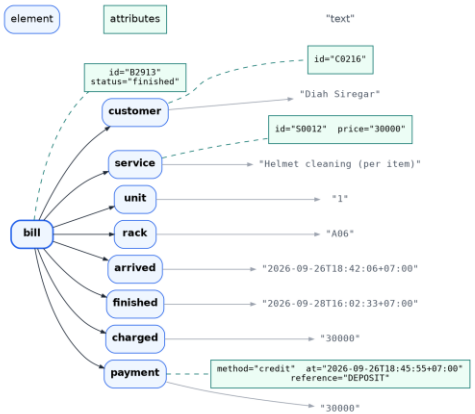


Figure 6.4 Bill B2913 as a tree. Elements are children of `bill`; attributes hang off the element that carries them; each leaf element has one text child.

***Figure 6.4**: attributes hang off their element; each leaf element has one text child. Book: Section 6.8.1.*

Worked Example 2: Counting the Nodes

Listing 6.10 The census that `python/04_minidom_nodes.py` prints for `bills.xml`, one line per kind of node.

```
attribute              31
comment                 1
document                1
element                38
text                   33
text, whitespace only  42
```

Elements: 1 `bills` + 4 `bill` + 9 + 9 + 7 + 8 = 33 children, so 1 + 4 + 33 = 38.

Whitespace-only text: 4 + 1 = 5 inside `bills`, and one more than the children inside each bill, 10 + 10 + 8 + 9 = 37: in all 5 + 37 = 42.

*In **Listing 6.10** the indentation that makes the file readable is content: `firstChild` of `<bill>` is a text node. Ask for children by name, or set `$doc->preserveWhiteSpace = false`. Book: Section 6.8.2.*

Namespaces

> **DEFINITION.** An XML **namespace** is a name for a vocabulary, written as a URI. An element's full name is the pair (namespace URI, local name).

Listing 6.11 From `data/bills-ns.xml`: two vocabularies, two prefixes, and two elements called `customer` that no longer collide.

```
<b:bills xmlns:b="https://laundry.profitina.example/ns/bill/1"
         xmlns:c="https://laundry.profitina.example/ns/customer/1"
         outlet="O01" date="2026-09-26">
  <b:bill id="B2913" status="finished">
    <b:customer id="C0216">Diah Siregar</b:customer>
    <c:customer id="C0216" gender="f">
      <c:name>Diah Siregar</c:name>
      <c:phone>0859-0000-7088</c:phone>
      <c:credit>62000</c:credit>
    </c:customer>
    <b:charged>30000</b:charged>
  </b:bill>
</b:bills>
```

*In **Listing 6.11** the prefix is a nickname: match on the URI. The URI is never fetched. An unprefixed attribute is in no namespace. Book: Section 6.9.1.*

The Famous Trap

Listing 6.12 The output of `php/09_namespaces.php`, which registers the two URIs under its own prefixes `x` and `k`.

```
//x:customer    -> 1 node: Diah Siregar
//k:customer    -> 1 node: phone 0859-0000-7088
//customer      -> 0 nodes
//*[local-name()='customer'] -> 2 nodes
$bill['id']              -> ''
$bill->attributes()->id  -> B2913
```

`//customer` matched zero nodes: an unprefixed name in XPath means *no namespace*. When an obviously right expression returns nothing, suspect a namespace first.

*In **Listing 6.12**, `$bill['id']` is empty because after `children(BILL)` SimpleXML looks for an `id` in that namespace; `attributes()` finds `B2913`. Book: Section 6.9.2.*

Well-Formed Is Not Valid

Listing 6.13 The first bill of `data/bills-invalid.xml`: every tag matches, every attribute is quoted, and it is wrong.

```
<?xml version="1.0" encoding="UTF-8"?>
<bills outlet="O01" date="26/09/2026">
  <bill id="2905" status="washing">
    <customer id="C0022">Vania Wirawan</customer>
    <service id="S0003" price="6000">Ironing only (per kg)</service>
    <unit>0</unit>
    <rack>B03</rack>
    <arrived>2026-09-26 09:05:04</arrived>
    <charged>42000</charged>
    <payment method="ovo" at="2026-09-27T14:43:55+07:00">42000</payment>
  </bill>
```

*The faults in **Listing 6.13**: the date `26/09/2026`, the id `2905`, the status `washing`, unit `0`, a time without `T` and zone, the method `ovo`; and the second bill puts `service` before `customer`. A parser accepts all seven. Book: Section 6.12.1.*

Validity and Grammars

> **DEFINITION.** A document is **valid** when, besides being well-formed, it obeys a stated **grammar**: which elements may appear, in what order, how often, with which attributes and, in XML Schema, what values. The grammar languages: the **DTD** and **XML Schema**.

A grammar is a check written **once**, **executable documentation** that cannot drift, and a failure moved **to the boundary**, with a line number.

Validation is about communication: two systems exchanging documents must speak the same language, written down in a form a machine can check. Book: Section 6.12.1.

The DTD: Sequence, Choice, How Many

Table 6.3 The four occurrence indicators of a DTD. They are the whole of counting in the DTD language.

Suffix	Meaning	Reading of `(name+, weight?, picture, subspecies*)`
none	exactly one	`picture` must appear once
`?`	zero or one	`weight` may appear once or not at all
`+`	one or more	`name` must appear at least once
`*`	zero or more	`subspecies` may appear any number of times, including none

> **EXAMPLE.** A bill needs a customer, service, unit, rack and arrival once each, then maybe a finish, maybe a collection, the charge, and any number of payments: `<!ELEMENT bill (customer, service, unit, rack, arrived, finished?, departed?, charged, payment*)>`.

*A comma is a sequence, `|` a choice, and the suffixes of **Table 6.3** count. B2911 is valid because `?` and `*` allow zero. Book: Sections 6.12.2 and 6.12.3.*

The Feed's DTD

Listing 6.14 From `data/bills.dtd`: the root, the content model of `bill`, and its three attributes.

```
<!ELEMENT bills     (bill*)>
<!ATTLIST bills     outlet CDATA #REQUIRED
                    date   CDATA #REQUIRED>

<!ELEMENT bill      (customer, service, unit, rack, arrived,
                     finished?, departed?, charged, payment*)>
<!ATTLIST bill      id        ID                          #REQUIRED
                    status    (queued|finished|departed) #REQUIRED
                    cancelled (true|false)               "false">
```

> **What a DTD cannot say:** that `unit` is a whole number from 1 to 999, that `arrived` is a real date and time, that an id is `B` and four digits. Of the seven faults it catches four.

*In **Listing 6.14**, `ID` must be unique and a legal XML name (so `2905` fails); `status` and `cancelled` are enumerations; `cancelled` defaults to `"false"`. Book: Sections 6.12.4 and 6.12.5.*

XML Schema: Values, Not Only Shapes

Table 6.4 The facets of XML Schema used in this chapter.

Facet	Constrains	Used in the feed for
`enumeration`	the value is one of a list	bill status, payment method
`pattern`	the value matches a regular expression	`B[0-9]{4}`, rack `[A-C][0-9]{2}`
`minInclusive` / `maxInclusive`	a numeric range, ends included	unit at most 999
`minExclusive` / `maxExclusive`	a numeric range, ends excluded	(not needed here)
`length` / `minLength` / `maxLength`	the number of characters	(the database's CHECK rules)

*An XSD is itself XML. Around forty built-in types (`xs:integer`, `xs:date`, `xs:dateTime`, …) are narrowed by the facets of **Table 6.4**; `minOccurs` and `maxOccurs` replace the DTD's suffixes. Book: Sections 6.13.1 and 6.13.2.*

Two Named Types from `bills.xsd`

Listing 6.15 From `data/bills.xsd`: a quantity is a positive integer at most 999, and a rack is a letter A to C and two digits.

```
<xs:simpleType name="quantity">        <!-- kg, items or pairs -->
  <xs:restriction base="xs:positiveInteger">
    <xs:maxInclusive value="999"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="rack">
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-C][0-9]{2}"/>
  </xs:restriction>
</xs:simpleType>
```

*The named simple types of **Listing 6.15** are defined once and used by name. `xs:simpleContent` with `xs:extension` says text plus attributes; `xs:unique` forbids two bills with one id. Book: Section 6.13.2.*

Worked Example 3: Seven Faults Found

Listing 6.16 What `php/04_validate.php` prints for `bills-invalid.xml` against `bills.xsd`: every fault, with its line.

```
bills-invalid.xml against bills.xsd  INVALID
    line 2: Element 'bills', attribute 'date': '26/09/2026' is not a valid value of the atomic type 'xs:date'.
    line 3: Element 'bill', attribute 'id': [facet 'pattern'] The value '2905' is not accepted by the pattern
'B[0-9]{4}'.
    line 3: Element 'bill', attribute 'id': Warning: No precomputed value available, the value was either invalid
or something strange happened.
    line 3: Element 'bill', attribute 'status': [facet 'enumeration'] The value 'washing' is not an element of the
set {'queued', 'finished', 'departed'}.
    line 6: Element 'unit': '0' is not a valid value of the atomic type 'quantity'.
    line 8: Element 'arrived': '2026-09-26 09:05:04' is not a valid value of the atomic type 'xs:dateTime'.
    line 10: Element 'payment', attribute 'method': [facet 'enumeration'] The value 'ovo' is not an element of the
set {'cash', 'qris', 'transfer', 'credit'}.
    line 13: Element 'service': This element is not expected. Expected is ( customer ).
```

> **KEY INSIGHT.** A schema checks grammar and datatypes, not the world. A valid bill can still be cancelled *and* charged; business rules are checked by the program, right after validation.

*The document parsed, so the validator of **Listing 6.16** kept going and reported everything in one run; the DTD found four. "No precomputed value" repeats the id complaint. Python's lxml prints the same messages; Java's `Validator` reports each fault twice. Validate at the boundary, once. Book: Section 6.13.3.*

DOM: Build the Whole Tree

Listing 6.17 From `main` in `java/BillsDom.java`: a factory that refuses any `DOCTYPE`, and the whole document parsed into a tree.

```
        DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
        // A feed from outside must not carry a DOCTYPE: this refuses both
        // entity attacks of Section 6.21 in one line.
        f.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
        Document doc = f.newDocumentBuilder().parse(new File("../data/bills.xml"));
        Element root = doc.getDocumentElement();
```

Listing 6.18 The day summary. `BillsDom.java`, `php/03_read.php` and `python/03_read_etree.py` print exactly this text; `diff` finds no difference.

```
Outlet O01, bills received on 2026-09-26
--------------------------------------------------------------------------
B2905  Vania Wirawan  Ironing only (per kg)      x7   departed   42.000  42.000
B2908  Laras Chandra  Wash, dry & fold (per kg)  x10  departed   80.000  80.000
B2911  Dimas Wibowo   Blanket (per item)         x1   cancelled       0       0
B2913  Diah Siregar   Helmet cleaning (per item) x1   finished   30.000  30.000
--------------------------------------------------------------------------
4 bills (1 cancelled), charged Rp 152.000, paid Rp 152.000
```

*After **Listing 6.17**, a loop over `getElementsByTagName("bill")` prints **Listing 6.18**; PHP and Python print exactly the same text. `getAttribute` returns `""` for an absent attribute. Book: Section 6.14.1.*

SAX: Events, One Pass, No Tree

Listing 6.19 `endElement` in `java/BillsSax.java`: the program keeps its own running totals, because SAX remembers nothing.

```
    @Override
    public void endElement(String uri, String local, String qName) {
        depth--;
        switch (qName) {
            case "charged" -> charged = Long.parseLong(text.toString().trim());
            case "payment" -> paid += Long.parseLong(text.toString().trim());
            case "bill" -> {
                System.out.printf("  %s  %-9s  charged %6d  paid %6d%n", billId, status, charged, paid);
                bills++;
                totalCharged += charged;
                totalPaid += paid;
            }
            default -> { }
        }
    }
```

*In **Listing 6.19** the parser calls `startElement`, `characters` and `endElement`; text is collected in a buffer because it may arrive in pieces. The totals are those of the day summary. Book: Section 6.14.2.*

Pull Parsing: a Cursor You Control

Listing 6.20 The loop of `java/BillsStax.java`: the program asks for the next event, and can `break`.

```
            while (r.hasNext()) {
                int event = r.next();
                if (event == XMLStreamConstants.START_ELEMENT) {
                    switch (r.getLocalName()) {
                        case "bill" -> System.out.printf("  bill %s (%s)%n",
                                r.getAttributeValue(null, "id"), r.getAttributeValue(null, "status"));
                        case "customer" -> System.out.printf("      customer %s: %s%n",
                                r.getAttributeValue(null, "id"), r.getElementText());
                        default -> { }
                    }
                } else if (event == XMLStreamConstants.END_ELEMENT
                        && r.getLocalName().equals("bill") && ++seen == wanted) {
                    System.out.println("  ...stopping: the rest of the file is never parsed.");
                    break;
                }
            }
```

Listing 6.20 *stops after the second `</bill>`; the rest of the file is never parsed. On four bills this saves nothing; on four million it is the difference between a second and an hour. Book: Section 6.14.3.*

Measuring the Difference

Listing 6.21 `total_streaming` in `python/09_stream_memory.py`: `iterparse`, and `el.clear()` to free each finished bill.

```python
def total_streaming():
    total = 0
    for event, el in ET.iterparse(YEAR, events=("end",)):
        if el.tag == "charged":
            total += int(el.text)
        elif el.tag == "bill":
            el.clear()          # free the finished bill, or nothing is saved
    return total
```

Listing 6.22 The output of **Listing 6.21**. The same total to the rupiah; the tree needed about thirteen times the memory.

```
bills-year.xml: 1.5 MB
ET.parse (whole tree)    total Rp 144.213.000   peak  13.4 MB
ET.iterparse + clear()   total Rp 144.213.000   peak   1.0 MB
```

***Listing 6.22**: a year of bills, 1.5 MB; the tree took 13.4 MB, streaming 1.0 MB, with the same total to the rupiah. Delete `el.clear()` and `iterparse` builds the whole tree anyway. Book: Section 6.14.4.*

Choosing a Reading Strategy

Table 6.5 The three reading strategies. The first row causes every other row.

	DOM	SAX	Pull (StAX, iterparse)
Builds a tree	yes	no	no
Memory	several times the file	small and constant	small and constant
Can look backwards	yes	no	no
Can stop early	pointless	no	yes
Can change the document	yes	no	no
Style of code	walk a structure	answer callbacks	drive a loop
Use for	files that fit in memory	huge files, one pass	huge files, early exit

***Table 6.5**: if the file fits comfortably in memory, use DOM, which is simpler and harder to get wrong. Budget about ten times the file for a tree. Book: Section 6.14.4.*

XPath: a Query Language for a Tree

DEFINITION. An **XPath expression** is a sequence of **steps** separated by `/`, each selecting nodes relative to the last, optionally narrowed by a **predicate** in `[ ]`. It may also compute a number, string or boolean.

Table 6.6 The XPath 1.0 syntax used in this chapter; enough for most real work.

Syntax	Meaning
`/bills`	the root element, if it is called `bills`
`/bills/bill`	every `bill` child of the root
`//payment`	every `payment` at any depth
`@id`	the `id` attribute
`..` and `.`	the parent, and the current node
`[ ... ]`	a predicate: keep only nodes for which it holds
`[3]`, `[last()]`	the third node (counting from **one**), the last node
`count()`, `sum()`, `contains()`, `not()`	functions

*XPath is to XML roughly what SQL is to a table, and because it is a separate standard, **Table 6.6** is the same in PHP, Java, Python, JavaScript and C#. Book: Section 6.15.1.*

Twelve Questions

Listing 6.23 The twelve expressions of `php/05_xpath.php`, each with the question it answers.

```
$queries = [
    '/bills/bill/@id'                       => 'every bill id, in document order',
    "//bill[@status='departed']/@id"        => 'a predicate [ ] keeps only some nodes',
    "//bill[@cancelled='true']/customer"    => 'who cancelled',
    '//bill[not(payment)]/@id'              => 'bills with no payment element at all',
    '//bill[finished and not(departed)]/@id'=> 'ready and waiting to be collected',
    '//bill[unit > 5]/customer'             => 'a comparison: more than 5 units',
    "//payment[@method='qris']/@reference"  => 'the QRIS reference numbers',
    "//service[contains(., '&')]"           => 'contains() on the text of the node',
    '//bill[last()]/@id'                    => 'last() is the final bill',
    'count(//bill)'                         => 'count() returns a number',
    'sum(//charged)'                        => 'sum() adds the values of a node set',
    'sum(//payment) div count(//payment)'   => 'div divides; / was already taken',
];
```

Listing 6.23 *selects bills by a property of a child (`unit > 5`), by an absence (`not(payment)`), and asks the counter's morning question: `finished and not(departed)`. Book: Section 6.15.2.*

Twelve Answers

Listing 6.24 Part of the output of `php/05_xpath.php`: an absence, a morning question, and a comparison.

```
//bill[not(payment)]/@id
    bills with no payment element at all
    -> id = B2911
//bill[finished and not(departed)]/@id
    ready and waiting to be collected
    -> id = B2913
//bill[unit > 5]/customer
    a comparison: more than 5 units
    -> customer = Vania Wirawan
    -> customer = Laras Chandra
count(//bill)
    count() returns a number
    -> 4.0
sum(//charged)
    sum() adds the values of a node set
    -> 152000.0
sum(//payment) div count(//payment)
    div divides; / was already taken
    -> 50666.666666666664
```

*`count()` returns `4.0` because XPath 1.0 has one number type, a double; `div` divides because `/` was taken. Java prints the same answers (**Listing 6.24**); Python's ElementTree knows only a subset, so use lxml. Book: Sections 6.15.2 and 6.15.3.*

XSLT: One Document into Another

DEFINITION. XSLT turns an XML document into HTML, text or other XML. A **stylesheet** is itself XML, run by an **XSLT processor**.

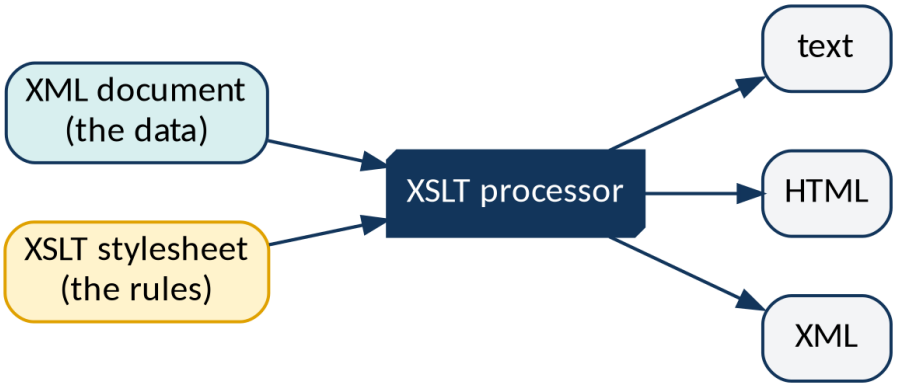


Figure 6.5 The XSL transformation. Two documents go in, the data and the stylesheet; one processor does the work; the output may be text, HTML or XML.

***Figure 6.5**: two inputs, one processor, three kinds of output. If the counter wants a phone column, the data does not change; only the stylesheet does. Book: Section 6.16.1.*

Template Rules

Table 6.7 The XSLT 1.0 instructions used in `day-sheet.xsl`.

Instruction	What it does
`xsl:template match="..."`	a rule: when a node matches this pattern, produce this
`xsl:apply-templates select="..."`	process these nodes, each with its own matching rule
`xsl:value-of select="..."`	insert the string value of an XPath expression
`xsl:for-each select="..."`	loop over nodes in place
`xsl:if test="..."`, `xsl:choose` / `xsl:when` / `xsl:otherwise`	conditions
`xsl:sort select="..."`	order the nodes that a loop or `apply-templates` visits
`xsl:variable name="..."`	name a value, which can never be changed
`document('...')` and `format-number()`	functions: read a second document; format a number

> **KEY INSIGHT.** XSLT is functional: an `xsl:variable` can never be reassigned. Totals come from `sum()`; the rest from templates that call themselves.

*A stylesheet is not a program with a beginning and an end: the processor walks the tree and applies the rule whose `match` fits each node (**Table 6.7**). Book: Section 6.16.2.*

The Day Sheet: One Row per Bill

Listing 6.25 Part of the `bill` template of `data/day-sheet.xsl`: `current()` joins the phone number from the customer document.

```
<xsl:template match="bill">
  <tr>
    <td><xsl:value-of select="@id"/></td>
    <td><xsl:value-of select="customer"/></td>
    <!-- current() is the bill being processed; without it, inside [ ] the
         expression customer/@id would be asked of a customer record. -->
    <td><xsl:value-of select="$people/customer[@id = current()/customer/@id]/phone"/></td>
    <td><xsl:value-of select="service"/> x <xsl:value-of select="unit"/></td>
    <td>
      <xsl:choose>
        <xsl:when test="@cancelled = 'true'">cancelled</xsl:when>
        <xsl:otherwise><xsl:value-of select="@status"/></xsl:otherwise>
      </xsl:choose>
    </td>
    <td><xsl:value-of select="format-number(charged, '#.##0', 'idr')"/></td>
```

*In **Listing 6.25**, `document('customers.xml')` loaded the second tree as `$people`; inside `[ ]` the context is a customer record, so `current()` is needed. The rows are sorted with `data-type="number"`, or `"9000"` follows `"80000"`. Book: Section 6.16.3.*

Running It Three Times

Listing 6.26 A row of the HTML that PHP's `XSLTProcessor` produced: B2908, the largest bill, first, with the phone number from the second document.

```
<tr>
<td>B2908</td>
<td>Laras Chandra</td>
<td>0877-0000-6820</td>
<td>Wash, dry &amp;amp; fold (per kg) x 10</td>
<td>departed</td>
<td>80.000</td>
<td>80.000</td>
<td>qris</td>
</tr>
```

*The output of **Listing 6.26** from Python's lxml is byte-identical (both use libxslt); Java's processor lays it out differently, yet the cells, order and totals agree. Browsers are removing XSLT (Chrome stops running it on November 17, 2026): transform on the server, which also keeps customer data private. Book: Sections 6.16.4 and 6.16.5.*

XML in the Browser

Listing 6.27 The processing instruction that attaches a stylesheet to an XML document.

```
<?xml-stylesheet type="text/xsl" href="day-sheet.xsl"?>
```

Listing 6.28 Parsing XML in the browser. `ids` becomes `["B2905", "B2908", "B2911", "B2913"]`.

```
const text = await (await fetch('/feed/bills.xml?outlet=O01&date=2026-09-26')).text();
const doc  = new DOMParser().parseFromString(text, 'application/xml');
const ids  = [...doc.querySelectorAll('bill')].map(b => b.getAttribute('id'));
```

*With **Listing 6.27** the browser transforms; with **Listing 6.28** your script does. The "X" in AJAX and `XMLHttpRequest` is XML: around 2005 the data fetched was XML, and JSON then took the job. Book: Section 6.17.*

Writing XML: Never by Joining Strings

> `"<service>" . $description . "</service>"` **breaks on the first real row,** *Wash, dry & fold*. It is Chapter 5's SQL-injection mistake in another language, with the same answer: use the library.

Listing 6.29 `addText` in `php/laundry_feed.php`: text goes in through `createTextNode`, which escapes `&` and `<`.

```
// Text goes in through createTextNode, never as createElement's second
// argument: only createTextNode escapes & and < for us.
function addText(DOMDocument $doc, DOMElement $parent, string $name, string $text): DOMElement
{
    $el = $doc->createElement($name);
    $el->appendChild($doc->createTextNode($text));
    $parent->appendChild($el);
    return $el;
}
```

*In **Listing 6.29**, `createElement('service', $text)` would **not** escape its second argument. Python uses ElementTree, Java StAX's `XMLStreamWriter`; none of the writers contains an angle bracket used as markup. Book: Section 6.18.*

Three Languages, Twelve Identical Files

Listing 6.30 What `compare_exports.py` prints after the PHP, Java and Python export programs have run.

```
bills.xml        1963 bytes   php same, java same, python same
bills.json       3292 bytes   php same, java same, python same
customers.xml    1325 bytes   php same, java same, python same
customers.json   1497 bytes   php same, java same, python same
```

***Listing 6.30**: three XML libraries and a hand-written JSON serializer agree to the byte. In general, compare canonical forms (Canonical XML, `c14n`), never raw bytes. The PHP export also validates its own output before handing it over. Book: Section 6.18.2.*

JSON from Zero

Table 6.8 The six JSON types. There are no comments, no attributes, no dates and no namespaces.

Type	Written as	Example from the feed
object	`{ "key": value, ... }`, keys are strings	a bill, a customer
array	`[ value, ... ]`, in order	`"payments": [ ... ]`
string	`"..."` in double quotes, with `\"`, `\\`, `\n`, `é` escapes	`"Wash, dry & fold (per kg)"`
number	decimal, optional fraction and exponent	`42000`
boolean	`true` or `false`	`"cancelled": false`
null	`null`	`"finished": null`

*JSON, standardized as ECMA-404 and RFC 8259, fits on a page and has exactly the six types of **Table 6.8**. Book: Section 6.19.1.*

The Cancelled Bill in JSON

Listing 6.31 Bill B2911 in `data/bills.json`: the same fields as the XML, with numbers, a boolean and a `null`.

```
            "id": "B2911",
            "status": "departed",
            "cancelled": true,
            "customer": {
                "id": "C0246",
                "name": "Dimas Wibowo"
            },
            "service": {
                "id": "S0009",
                "description": "Blanket (per item)",
                "price": 25000
            },
            "unit": 1,
            "rack": "A17",
            "arrived": "2026-09-26T17:06:02+07:00",
            "finished": null,
            "departed": "2026-09-27T07:46:02+07:00",
            "charged": 0,
            "payments": []
```

***Gained** in **Listing 6.31**: numbers and booleans in the data, and a direct fit with every language's objects. **Lost:** the comment, the attribute-or-element distinction, and dates, which are strings by agreement. Book: Section 6.19.1.*

Types, Absence, Size

Listing 6.32 Part of the output of `php/07_json.php`: the same values read from both formats, and the sizes of the same four bills.

```
The same value, read from each format:
  B2905 charged    XML: string '42000'   JSON: int 42000
  B2905 cancelled  XML: absent           JSON: bool false
  B2911 finished   XML: string ''        JSON: null
  isset($xml->bill[2]->finished) is false

Sizes of the same four bills:
  XML as written (indented, commented)    1963 bytes
  JSON as written (pretty-printed)        3292 bytes
  XML squeezed                            1641 bytes
  JSON compact                            1654 bytes
  XML squeezed, gzip                       614 bytes
  JSON compact, gzip                       588 bytes
```

*In **Listing 6.32**, from XML `charged` is a string, from JSON an integer. Absent (XML) and `null` (JSON) are different documents: this feed writes every key. Squeezed and gzipped, the formats are within a few bytes: size is not the argument. Book: Sections 6.19.2 and 6.19.3.*

XML Against JSON

Table 6.9 XML against JSON. The upper rows are why XML survives where it survives; the last row is why JSON took the web.

	XML	JSON
Types in the data	text only (types come from the schema)	string, number, boolean, null, object, array
Comments	yes	no
Attributes as well as content	yes	no
Mixed content (text with markup inside)	yes	no
Namespaces	yes	no
Grammar language	DTD, XML Schema (mature)	JSON Schema (widely used, less mature)
Query language	XPath (one standard)	JSONPath (RFC 9535, 2024), JMESPath, jq
Transformation language	XSLT (standard)	none standard
Native in the browser	`DOMParser`	`JSON.parse`, `response.json()`

*JSON took the web because it is JavaScript's own syntax, maps onto records, carries types, is tiny to implement, and came with REST. XML keeps marked-up text, grammars several parties agreed on, and standard transformation (**Table 6.9**). Book: Section 6.19.4.*

A Feed Endpoint

Addresses: `/feed/bills.xml` and `/feed/bills.json` with `?outlet=O01&date=2026-09-26`; without an extension, the `Accept` header chooses, and the response says `Vary: Accept`.

Media types: `application/xml` and `application/json`, both `charset=utf-8`. **Caching:** `private, max-age=60`, because the feed holds customer names.

Listing 6.33 From `php/feed.php`: both query parameters are checked before the database is touched.

```
// Never trust a query string: check both parameters before touching the database.
$outlet = strtoupper($_GET['outlet'] ?? '');
$date   = $_GET['date'] ?? '';
$day    = DateTime::createFromFormat('!Y-m-d', $date);
if (!preg_match('/^O[0-9]{2}$/', $outlet)) {
    fail(400, 'bad_request', 'outlet must look like O01.');
}
if ($day === false || $day->format('Y-m-d') !== $date) {
    fail(400, 'bad_request', 'date must be a real date, YYYY-MM-DD.');
}
```

*In **Listing 6.33** `2026-02-30` fails the round-trip check and gets `400 Bad Request` with the Laundry API's error body. Flask and the JDK's `HttpServer` implement the same design and return byte-identical bodies. Book: Section 6.20.*

The Entity Bomb

Listing 6.34 `data/bomb.xml`, the entity-expansion bomb known as *billion laughs*. Each entity is ten copies of the one below it, so `lol4` is 10^4 copies of `lol`: 30,000 characters from 411 bytes. Five more levels would make three billion.

```
<?xml version="1.0"?>
<!DOCTYPE bills [
  <!ENTITY lol  "lol">
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
]>
<bills><note>&lol4;</note></bills>
```

Listing 6.35 Java's answer, from `java/EntityTraps.java`: the default factory expands it; the hardened one refuses any `DOCTYPE`.

```
bomb.xml  defaults  accepted; note holds 30000 characters: lollollollollollollollollollo
bomb.xml  hardened  refused: DOCTYPE is disallowed
```

*PHP's libxml refuses **Listing 6.34**; Python expands it; **Listing 6.35** shows the JDK expanding it until the factory refuses any `DOCTYPE`. Measure your own stack's defaults instead of assuming them. Book: Sections 6.21.1 and 6.21.2.*

XXE: the Document That Reads Your Files

Listing 6.36 `data/xxe.xml`: `SYSTEM` asks the parser to fetch a file on the server.

```
<?xml version="1.0"?>
<!DOCTYPE bills [
  <!ENTITY leak SYSTEM "secret.txt">
]>
<bills><note>&leak;</note></bills>
```

Listing 6.37 Java: **the JDK's default `DocumentBuilderFactory` read the file.**

```
xxe.xml   defaults  accepted; note holds 32 characters: DB_PASSWORD=not-a-real-password
xxe.xml   hardened  refused: DOCTYPE is disallowed
```

> **PHP:** never pass `LIBXML_NOENT`. **Java:** `disallow-doctype-decl` on every factory. **Python:** keep lxml's `resolve_entities` off. **Everywhere:** a size and time limit.

*With **Listing 6.36**, PHP's defaults read nothing; **Listing 6.37** shows the JDK's default factory reading `secret.txt`. A feed never needs a `DOCTYPE`. Book: Section 6.21.3.*

Profitina in Practice

Structured data is what a machine can cite. Profitina (`app.profitina.com`) reports on schema.org facts embedded as **JSON-LD**: Section 6.9's namespace idea in JSON clothing.

A plain file for answer engines: `/llms.txt`, short Markdown that tells language models what a site is.

Discovery runs on XML: `sitemap.xml`, one `<url>` per page; reading it is the DOM loop with other names.

Validate at the boundary: the scanner reads documents others wrote, with size and time limits and external entities off.

The app advises; the user decides. Runnable practice is always Profitina Laundry. Book: Section 6.22.

Common Mistakes

Writing: joining strings; `createElement` with text in PHP; an empty element for a missing value; the database's time format in an `xs:dateTime`.

Reading: only the last error message; `firstChild` expected to be an element; forgetting to cast in PHP, or that `.text` may be `None`.

Querying: a namespaced document without registering the namespace; numbers sorted as text in XSLT; a forgotten `current()`.

Safety and scale: DOM on a file that does not fit, `iterparse` without `clear()`; parser entity defaults left unchecked; absent confused with `null`.

Each mistake is listed in the book with its fix. Book: Section 6.23.

Summary: Five Things to Remember

1. XML lets strangers' programs exchange data because names travel with values; it is a meta-language with six rules of well-formedness.

2. A document is a tree of nodes, whitespace included; namespaces make a name the pair (URI, local name).

3. Well-formed is not valid: a DTD states shapes, an XML Schema shapes and values. Validate at the boundary, then check business rules.

4. Read with DOM, SAX or a pull parser; ask with XPath; transform with XSLT; write only through a library.

5. JSON has six types and won the web; a feed serves both, validates its parameters, and treats every document from a stranger as untrusted input.

The twelve exported files agree to the byte in PHP, Java and Python. Book: Section 6.24.

Next Time

Chapter 7 — Java Server Pages. Servlets and containers, the JSP lifecycle, scopes, JavaBeans, EL and JSTL, filters, and JDBC in a web application.

Chapter 7 edits a `web.xml` by hand: the XML of today configures the server of next time.

Before then: work Exercises 6.2, 6.5, 6.8, 6.13 and 6.14.

Solutions to every exercise are in the book. Book: Sections 6.25 and 6.26.

Lecture 7 — Java Server Pages

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 7

Lecturer: Irfan Subakti — yifana@gmail.com

Today the box in the middle speaks Java: a container, servlets, and pages that become servlets. Book: Chapter 7, Sections 7.1–7.26.

What This Meeting Covers

1. **The container** — why server-side Java exists, servlets, `javax` against `jakarta`, getting Tomcat running.
2. **What happens to a JSP page** — translation, compilation, the generated class, the lifecycle.
3. **Putting Java in a page** — directives, scripting elements, the nine implicit objects, the four scopes.
4. **Taking the Java out again** — JavaBeans, the Expression Language, JSTL and `<c:out>`.
5. **The Laundry bills pages** — MVC, JDBC through a DAO, login and sessions, filters, errors; the same pages on the JDK's server and in Flask.

Everything invented after the first JSP exists to take the Java back out of the page; the order of today follows that story. Book: Section 7.1.

Where We Left Off

A browser asks, a server answers (Chapter 1); HTML and CSS fill the answer (Chapter 2); PHP computes it (Chapter 3); MySQL remembers (Chapter 5); XML carries data between programs (Chapter 6).

One thing changes today: the language in the box in the middle. “The PHP interpreter” becomes “a Java servlet container”.

Three questions: why does this exist at all? What actually happens to a JSP page? How do I keep the Java out of the page?

The vocabulary is heavy (servlet, container, WAR, filter, bean, EL, tag library) and hides how little is new: a request comes in, some code runs, HTML goes out. Book: Section 7.1.

Why Server-Side Java, and What a Container Is

THE KEY INSIGHT. Server-side Java won because the business logic (payroll, inventory, accounting) was already in Java, and the shortest path from a browser to it was a Java program.

DEFINITION. A **container** runs your code and supplies everything around it: the socket, the threads, HTTP parsing, the lifecycle of your objects, pooling and naming. Apache **Tomcat** is the container of this chapter.

*In PHP you write a file and the file runs; in Java you write a class and hand it over, and the container decides when it runs. That **inversion of control** is why Java needs configuration. Book: Sections 7.2.1 and 7.2.2.*

One Request Through the Container

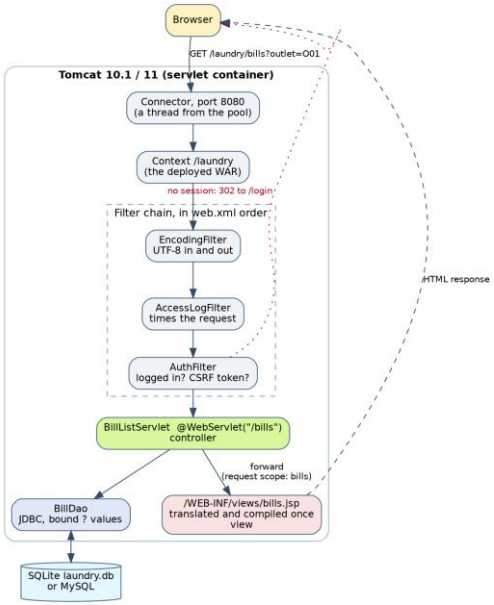


Figure 7.1 One request through the servlet container, as this chapter's Laundry application handles it. The filters run in the order `web.xml` gives them; the servlet decides; the JSP page draws. A request without a session is turned back by the third filter before any servlet runs.

***Figure 7.1**: a connector on port 8080, the application at `/laundry`, three filters in the order of `web.xml`, a servlet that decides, and a JSP page that draws. Every arrow is Tomcat calling our code. Book: Section 7.2.2.*

A Servlet

Listing 7.1 The class in `HelloServlet.java`: `@WebServlet("/hello")`, `extends HttpServlet`, and a `doGet` that prints the page line by line.

```
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        resp.setContentType("text/html; charset=UTF-8");
        PrintWriter out = resp.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("Hello, Web Programming class!");
        out.println("<br>");
        out.println("Current time is " + new java.util.Date());
        out.println("</body>");
        out.println("</html>");
    }
```

*A **servlet** is a Java class the container calls for an address; it receives a request object and a response object. In **Listing 7.1** every character of HTML is a Java string: invisible to editors, full of escaped quotes, and every change needs a compile and a redeploy. Book: Section 7.2.3.*

Turning the Page Inside Out

Table 7.1 Servlets and JSP compared. The third row is the one to remember: a JSP page does not replace the servlet, it *becomes* one. The last row is why real applications use both.

	Servlets	JSP
What it looks like	HTML code inside Java	Java-like code inside HTML
Ease of writing pages	Not easy	Very easy
What it is, underneath	A Java class you write and compile	A page that is compiled into a servlet
Who edits it comfortably	A Java developer	A Java developer, and a page designer
What it is best at	Controlling and dispatching	Displaying

> **THE KEY INSIGHT.** JSP is not a second technology beside servlets. It is a way of *writing* a servlet in which the HTML is written as HTML.

*The third row of **Table 7.1**: a JSP page does not replace the servlet, it becomes one. The standing advice: minimize the scripting elements in pages. Book: Section 7.2.4.*

`javax` and `jakarta`

Table 7.2 Tomcat versions and the specifications they implement. This chapter's code is written to Jakarta Servlet 6.0 and Pages 3.1, and so runs on Tomcat 10.1 and on Tomcat 11, which accepts everything 10.1 accepts.

Tomcat	Servlet	Pages (JSP)	EL	Package	Needs Java	Status, September 2026
8.5	3.1	2.3	3.0	`javax`	7+	end of life (March 2024); last release 8.5.100
9.0	4.0	2.3	3.0	`javax`	8+	legacy, supported to March 2027 (9.0.122)
10.1	**6.0**	**3.1**	5.0	`jakarta`	11+	current (10.1.60)
11.0	6.1	4.0	6.0	`jakarta`	17+	current, newest (11.0.26)

> **A `javax` WAR on a `jakarta` container does nothing useful.** Tomcat 10 and 11 find no class extending `jakarta.servlet.http.HttpServlet` and answer every address with 404.

*Rules from **Table 7.2**: use Tomcat 10.1 or 11; import `jakarta.servlet…`; fix old examples by replacing the package prefix. `java.sql`, `javax.sql` and `javax.naming` belong to Java itself and keep their names. Book: Section 7.2.5.*

Getting Tomcat Running

A JDK, not just a runtime: `java -version` and `javac -version` must both answer; set `JAVA_HOME`.

Unpacking is installing: download Tomcat 11 (or 10.1) from `tomcat.apache.org`, unzip to `CATALINA_HOME`, run `catalina.bat run` or `bin/catalina.sh run`; on macOS `brew install tomcat`.

Prove it: `http://localhost:8080/`. The port must be written; Apache and Tomcat are different servers.

In VS Code: *Community Server Connectors* creates, starts, deploys to and stops Tomcat from the side bar.

> **Never ship `admin`/`admin` in `tomcat-users.xml`.** The manager accepts an uploaded `.war`, and a `.war` is a program.

XAMPP 8.2.12 bundles Tomcat 8.5.96, a `javax` container past its end of life: fine for the first page, not for this chapter's servlets. Book: Sections 7.3.1–7.3.6.

The First JSP Page

Listing 7.2 `hello.jsp`, the first JSP page. Everything is ordinary HTML except the characters from `<%=` to `%>`.

```
<html>
    <body>
        Hello, Web Programming class!
        <br>
        Current time is <%= new java.util.Date() %>
    </body>
</html>
```

Listing 7.3 The real response for **Listing 7.2**, served by MiniJSP. Every character of template text survived, including the tab characters, and the expression became a date.

```
<html>
    <body>
        Hello, Web Programming class!
        <br>
        Current time is Wed Sep 23 16:40:25 ICT 2026
    </body>
</html>
```

***Template text** in **Listing 7.2** is copied unchanged; `<%= … %>` is a hole filled at request time. In **Listing 7.3** the tag has vanished, the whitespace survived, and the date is the server's. MiniJSP, a small engine that needs only a JDK, served it. Book: Section 7.4.*

Worked Example 1: Predict the Output

WORKED EXAMPLE. Predict `Your bill is <%= 4 * 12000 %> rupiah.`

Step 1. `Your bill is` is template text: copied.

Step 2. `4 * 12000` is Java integer arithmetic: $4 \times 12000 = 48000$, converted to the text `48000`.

Step 3. `rupiah.` is copied. The browser receives `Your bill is 48000 rupiah.`

What is missing: no separator, no `Rp`. A bean's `getPaymentRupiah()` returns `Rp 48.000` instead.

And `<%= 4 * 12000; %>`? The generated line is `out.print(4 * 12000);`, which does not compile. Book: Sections 7.4.3 and 7.5.3.

A JSP Page Is a Servlet in Disguise

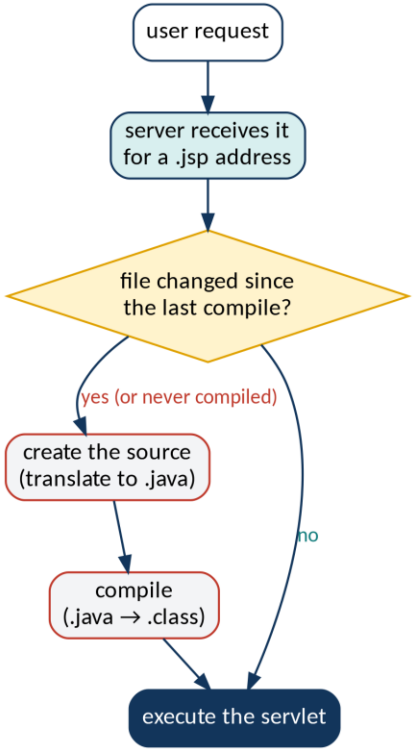


Figure 7.2 How a JSP page is served. The server asks one question — *has the file changed?* If it has, the source is created and compiled; if not, the already compiled servlet is executed at once. The expensive path is taken once; the cheap path is taken ever after.

***Figure 7.2**: the page is **translated** into Java, **compiled**, loaded and instantiated once, and only when the file has changed. The first request is slow, editing a live page just works, and a container needs a compiler at run time. Book: Sections 7.5.1 and 7.5.2.*

The Generated Servlet

Listing 7.4 The body of `_jspService` in the class MiniJSP generated from `hello.jsp`: the implicit objects declared, `out.write` for template text, `out.print` for the expression.

```
        HttpSession     session     = pageContext.getSession();
        ServletContext  application = pageContext.getServletContext();
        ServletConfig   config      = pageContext.getServletConfig();
        JspWriter       out         = pageContext.getOut();
        response.setContentType("text/html; charset=UTF-8");

        out.write("<html>\n");
        out.write("\t<body>\n");
        out.write("\t\tHello, Web Programming class!\n");
        out.write("\t\t<br>\n");
        out.write("\t\tCurrent time is ");
        out.print(new java.util.Date());
        out.write("\n");
        out.write("\t</body>\n");
        out.write("</html>\n");
```

*In **Listing 7.4** the class `hello_jsp extends JspPage`; `_jspService` runs once per request; the variables at the top are the implicit objects. A compile error names the generated file: read the line "in the jsp file". Book: Sections 7.5.3 and 7.5.4.*

The Life of One Request

Table 7.3 The life of one JSP request. Steps 7 and 8 happen once for the life of the page; the others happen for every request.

Step	Who does it	What happens
1	Browser	A link is clicked or a form is submitted
2	Browser	An HTTP request is built: method, path, headers, sometimes a body
3	Network	The request reaches the server named in the address
4	Tomcat	The connector accepts it and gives it to a thread from the pool
5	Tomcat	The path is matched to an application, then to a servlet or page
6	Tomcat	The filters mapped to that path run, in order (Section 7.18)
7	JSP engine	Has the `.jsp` file changed? If so, translate and compile it (**Figure 7.2**)
8	JSP engine	If this is the first use, construct the page object and call `jspInit()`
9	Page	`_jspService()` runs once, on one thread, writing into `out`
10	Tomcat	The buffer is flushed: status line, headers, then the body
11	Network, browser	The response travels back; the browser parses and draws it
—	Tomcat	Much later, at shutdown or reload, `jspDestroy()` is called once

*Steps 7 and 8 of **Table 7.3** happen once for the life of the page: **one instance serves every visitor**, and step 9 runs on many threads at once. Local variables are safe; fields are not. Book: Section 7.5.5.*

A Page That Counts

> **DEFINITION.** The **JSP lifecycle**: `jspInit()` once, `_jspService()` once per request, `jspDestroy()` once. You may write the first and the last, never the middle one.

Listing 7.5 The declaration in `counter.jsp`: a field, a synchronized method that increments it, and the two lifecycle methods.

```
<%!
    // A declaration. This is a field of the generated servlet class, not a
    // local variable, so it survives from one request to the next -- and it
    // is shared by every visitor at once, which is why it must be guarded.
    private int hits = 0;

    private synchronized int nextHit() { return ++hits; }

    public void jspInit()    { System.out.println("counter.jsp: jspInit()"); }
    public void jspDestroy() { System.out.println("counter.jsp: jspDestroy()"); }
```

*Everything in **Listing 7.5** becomes a member of the generated class, outside every method. Further down, the page writes `<% int local = 0; local++; %>`, a local variable of `_jspService`. Book: Section 7.5.6.*

Field or Local Variable?

Listing 7.6 The real output of two consecutive requests to **Listing 7.5**. The field counts up; the local variable does not; the random heading is new each time because the *method* is shared but each call computes afresh.

```
first request:   <p>This page has been served 1 time(s) since it was compiled.</p>
                 <h3>0.6947807836386516</h3>
                 <p>Local variable: 1.</p>
second request:  <p>This page has been served 2 time(s) since it was compiled.</p>
                 <h3>0.05855877850633684</h3>
                 <p>Local variable: 1.</p>
```

> **The nastiest bug of the chapter.** A field declared in `<%! … %>` is shared by every visitor at once. `<%! private String customerName; %>` set from the request lets two cashiers see each other's customer.

*In **Listing 7.6** the field counts 1, 2; the local variable is 1 every time; `jspInit()` printed once on the console. Never put request data in a declaration. Book: Section 7.5.6.*

Servlets in Full

Lifecycle: one instance, `init()` once, `service()` per request on many threads, `destroy()` at the end; `doGet`, `doPost`; an unimplemented method answers 405.

Request: `getParameter`, `getParameterValues`, `getHeader`, `getSession`, `setAttribute`, `getRequestDispatcher`. **Response:** `setContentType`, `getWriter`, `setStatus`, `sendError`, `sendRedirect`, `addCookie`.

Mapping: `@WebServlet("/bills")` or `<servlet-mapping>` in `web.xml`; patterns exact (`/bills`), prefix (`/bill/*`), extension (`*.jsp`). The descriptor wins.

A servlet's fields may hold only what is the same for every request; the Laundry servlets have none. Book: Sections 7.6.1–7.6.4.

A Web Application: `WEB-INF` and the WAR

> **DEFINITION.** `WEB-INF` is never served to a browser: classes, jars, `web.xml` and the views live there. A **WAR** is a ZIP of the application in that layout, the unit of deployment. The **deployment descriptor** `WEB-INF/web.xml` holds settings, mappings, error pages and session rules, and no logic.

Six steps: compile; static resources; `web.xml`; build the WAR with the JDK alone; drop it into `webapps/` (or the manager, or VS Code); ask for the page.

> **Never put the Servlet API inside your WAR.** The container supplies it; a second copy gives a class that cannot be cast to itself. JSTL, however, must be carried in `WEB-INF/lib`.

The Laundry views live in `WEB-INF/views/`, reachable only by a forward from a servlet, so no side entrance skips the login check. Book: Sections 7.7.1–7.7.4.

Filters in `web.xml`, in Order

Listing 7.7 The filter mappings of `WEB-INF/web.xml`: encoding and access log for every address, the login check for the bill pages only.

```
  <filter-mapping>
    <filter-name>encoding</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>accessLog</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>auth</filter-name>
    <url-pattern>/bills</url-pattern>
    <url-pattern>/bills/*</url-pattern>
    <url-pattern>/bill</url-pattern>
    <url-pattern>/bill/*</url-pattern>
  </filter-mapping>
```

*Only `web.xml` fixes the **order** of filters, which is why **Listing 7.7** declares them there. The same file sets `shopName`, the database URL, the error pages and a 30-minute `HttpOnly` session. Book: Section 7.7.2.*

Worked Example 2: Which Address?

> **WORKED EXAMPLE.** The WAR is `laundry.war`, `BillListServlet` says `@WebServlet("/bills")`, Tomcat runs on port 8080. What address reaches the servlet, and what is wrong with `<a href="/bills">`?

Server `http://localhost:8080` + **context path** `/laundry` (from the file name) + **pattern** `/bills` = `http://localhost:8080/laundry/bills`.

`<a href="/bills">` points to `http://localhost:8080/bills`, without the context path: **404**.

Every Laundry link is written `${ctx}/bills`, with `ctx` set once to `${pageContext.request.contextPath}`; rename the WAR and the links still work.

A servlet reads a query value as `request.getParameter("username")`, a page as `${param.username}`. Book: Section 7.7.5.

Directives

> **DEFINITION.** A **directive**, `<%@ … %>`, is a message to the translator about the servlet it will generate. It produces no output and is obeyed once, in the translation phase.

`page`: `import`, `contentType="text/html; charset=UTF-8"`, `errorPage`, `isErrorPage`, `session`. `isThreadSafe="true"` is *your* promise, not a service.

`include`: `<%@ include file="/WEB-INF/jspf/header.jspf" %>` pastes the *text* before translation: afterwards there is one page, sharing variables.

`taglib`: `<%@ taglib prefix="c" uri="jakarta.tags.core" %>`; then `<c:forEach>` is the `forEach` tag of the library called `c`.

Every Laundry view includes `header.jspf`, which declares the tag libraries and sets `ctx`. Book: Sections 7.8.1–7.8.5.

Scripting Elements

Table 7.4 The three scripting elements and the JSP comment. A declaration is not in the method, which is why it behaves so differently from the other two.

Form	Written as	Where the code lands	Ends with a semicolon?
Expression	`<%= expr %>`	Inside `_jspService`, as `out.print(expr);`	**No**
Scriptlet	`<% code %>`	Inside `_jspService`, verbatim	Yes, every statement
Declaration	`<%! code %>`	Inside the **class**, outside every method	Yes
Comment	`<%-- text --%>`	Nowhere at all	—

> `<%= request.getParameter("x") %>` **is Chapter 3's cross-site scripting hole, in Java.** The cure is still: escape on output, in JSP with `<c:out>`.

*The third column of **Table 7.4** explains everything: an expression becomes `out.print(expr);`, a scriptlet is pasted into the method, a declaration lands in the class and cannot see `request`. Book: Sections 7.9.1–7.9.5.*

A Scriptlet That Spans HTML

Listing 7.8 The shopping-cart loop. The `while` opens in the first scriptlet and closes in the last; the template text and expressions between them are the loop body, written once per item.

```
<%
    Iterator i = cart.getItems().iterator();
    while (i.hasNext()) {
        ShoppingCartItem item = (ShoppingCartItem) i.next();
        BookDetails bd = (BookDetails) item.getItem();
%>
        <tr>
            <td align="right" bgcolor="#ffffff">
                <%= item.getQuantity() %>
            </td>
            <td bgcolor="#ffffaa">
                <strong><a href="<%= request.getContextPath() %>/bookdetails?bookId=<%= bd.getBookId() %>"><%=
bd.getTitle() %></a></strong>
            </td>
        </tr>
<%
    // End of while
    }
%>
```

*In **Listing 7.8** the `while` opens in one scriptlet and closes in another. With two items in the cart, `hasNext()` is true twice and two rows are written; their prices* $39.95 + 39.99 = 79.94$*. Book: Section 7.9.3.*

What the Loop Becomes

Listing 7.9 What **Listing 7.8** translates into, in outline. The HTML between the scriptlets became `out.write` calls inside the `while` body. There was never anything clever going on.

```
Iterator i = cart.getItems().iterator();
while (i.hasNext()) {
    ShoppingCartItem item = (ShoppingCartItem) i.next();
    BookDetails bd = (BookDetails) item.getItem();
    out.write("\t\t<tr>\n");
    out.write("\t\t\t<td align=\"right\" bgcolor=\"#ffffff\">\n");
    out.print(item.getQuantity());
    out.write("\n\t\t\t</td>\n");
    // ... the other cells, written the same way ...
    out.write("\t\t</tr>\n");
    // End of while
}
```

> **The unclosed brace.** Delete the last scriptlet and the compiler reports "insert `}` to complete Block" at the end of a generated file. `<c:forEach>` cannot be left open unnoticed.

***Listing 7.9**: the HTML between the scriptlets became `out.write` calls inside the `while` body. There was never anything clever going on. Book: Section 7.9.3.*

The Implicit Objects

Table 7.5 Seven of the nine implicit objects; the other two are `config`, this page's own settings, and `page`, which is `this`.

Name	Type	Lives for	Most used for
`request`	`HttpServletRequest`	this request	`getParameter`, `getHeader`, `setAttribute`
`response`	`HttpServletResponse`	this request	`setContentType`, `addCookie`, `sendRedirect`
`out`	`JspWriter`	this request	`print`, `write`
`session`	`HttpSession`	this browser	remembering across requests
`application`	`ServletContext`	the whole application	shared data, `getInitParameter`, `log`
`pageContext`	`PageContext`	this request	reaching all four scopes at once
`exception`	`Throwable`	this request	only on a page with `isErrorPage="true"`

***Table 7.5**: local variables declared by the translator. `getParameter` returns `null` for a missing field; the Laundry reads parameters through `Validator.field`. Book: Section 7.10.*

The Four Scopes

DEFINITION. A **scope** is a named map from strings to objects, with a lifetime: **page**, **request** (including forwards), **session** (this browser) and **application** (everyone, until the application stops).

Table 7.6 The four scopes. Reading down the "until" column is reading the four answers to "how long should this live?"

Scope	Held by	Lives from	Until	In the Laundry application
page	`pageContext`	the start of this page	the end of this page	`ctx`, the loop variable `b`
request	`request`	the request arriving	the response being sent	`bills`, `bill`, `form`, `errors`
session	`session`	the first request from a browser	timeout, logout, browser closed	`user`, `csrf`, `flash`
application	`application`	deployment	undeployment or shutdown	`billDao`, `services`, `outlets`

Table 7.6 *answers "how long should this live?"* `${name}` *searches page, request, session, application, and the narrowest wins; name the scope when it matters:* `${sessionScope.user}`*. Book: Sections 7.11.1 and 7.11.3.*

Watching the Scopes Move

Listing 7.10 The real output of two requests to `scopes.jsp`. The page counter never moves; the request stamp is new each time and remembers nothing; the session and application counters climb together.

```
first request                    second request
page          1                  page          1
request       1790156426002      request       1790156426036
session       1                  session       2
application   1                  application   2
```

> **Application scope is shared by everyone, at the same instant.** Two requests can both read 7 and write 8. Keep read-mostly things there; put counters in the database.

*In **Listing 7.10** page never moves, request is new each time, session and application climb together; from a second browser the session starts again at 1 while application continues. Book: Section 7.11.2.*

JavaBeans: a Page with No Java

> **DEFINITION.** A **JavaBean** has a public no-argument constructor, private fields, and a property for each public `getX()` (or `isX()`) and `setX(value)`. `getPaymentRupiah()` makes `paymentRupiah` a property with no such field.

Listing 7.11 The loop in MiniJSP's `bills.jsp`: EL reads the bean's properties, JSTL repeats, `<c:out>` escapes.

```
      <c:forEach var="b" items="${book.bills}" varStatus="st">
        <tr>
          <td>${st.count}</td>
          <td>${b.id}</td>
          <td><c:out value="${b.customerName}"/></td>
          <td><c:out value="${b.serviceDescription}"/><c:if test="${b.serviceId eq 'S0002'}">
(express)</c:if></td>
          <td>${b.quantity}</td>
          <td>${b.paymentRupiah}</td>
        </tr>
      </c:forEach>
```

*Above **Listing 7.11**, `<jsp:useBean id="book" class="laundry.BillBook"/>` finds or creates the bean and `<jsp:setProperty … param="outlet"/>` sets one property. Never `property="*"` on a bean with a price: the visitor would choose it. Book: Sections 7.12.1–7.12.3.*

Worked Example 3: Checking the Queue

Listing 7.12 The real output of `bills.jsp?outlet=O01` against the sample database. Without `?outlet=`, the same page listed all 23 queued bills of the three outlets, Rp 1.003.000 in total.

```
#  Bill   Customer          Service                                Qty     Charged
1  B2953  Lestari Mahendra  Dry cleaning - dress/kebaya (per item)  1 item  Rp 45.000
2  B2949  Zahra Halim       Wash, dry & fold (per kg)              7 kg    Rp 56.000
3  B2948  Joko Mahendra     Ironing only (per kg)                  6 kg    Rp 36.000
…
9 bill(s) waiting, Rp 384.000 charged in total.
```

WORKED EXAMPLE. B2949: wash, dry and fold, Rp 8.000 per kg, 7 kg: $7 \times 8000 = 56000$, printed `Rp 56.000`. B2948: ironing, Rp 6.000 per kg, 6 kg: $6 \times 6000 = 36000$, printed `Rp 36.000`.

*The bean did no arithmetic in **Listing 7.12**: the amount was fixed when each bill was created, so a later price change never rewrites history. Book: Section 7.12.3.*

The Expression Language

Listing 7.13 The same value twice. The first needs a cast, a fully qualified class name, a method call and three kinds of bracket. The second needs none of them and does not break when the class moves to another package.

```
<%= ((laundry.model.Bill) request.getAttribute("bill")).getCustomerName() %>

${bill.customerName}
```

`${ … }` reads properties, walks into maps and lists, compares and tests for emptiness, and deliberately cannot query a database or assign.

The dot walks in: `${bill.customerName}` calls `getCustomerName()`. **Brackets** allow other keys: `${prices['wash-fold']}`. **A missing name prints nothing.**

EL's own names: `param`, `paramValues`, `header`, `cookie`, `initParam`, the four `…Scope` maps, and `pageContext`.

*The first line of **Listing 7.13** needs a cast, a class name and a method call; the second needs none and survives a package move. Book: Sections 7.13.1, 7.13.2 and 7.13.4.*

Every Operator, with Its Answer

Listing 7.14 The real output of `el-demo.jsp`. The numeric rows follow the EL rules for types, which the notes below explain.

```
${kilos}                                                  3.2
${kilos * price}                                          38400.0
${kilos gt 3}                                             true
${kilos gt 3 ? 'over three kilos' : 'three kilos or less'}  over three kilos
${empty customer}                                         false
${empty missingName}                                      true
${missingName}                                            (nothing at all)
${param.phone}                                            0896-0000-9434
${prices['wash-fold']}                                    8000
${prices.wash-fold}                                       0
${header['user-agent']}                                   curl/8.5.0
${pageContext.request.method}                             GET
${(2 + 3) * 4}                                            20
${10 div 4}                                               2.5
${10 mod 4}                                               2
```

*In **Listing 7.14** whole numbers stay whole under `+`, `-`, `*`, `mod`; `/` and `div` give floating point; `${prices.wash-fold}` is `0` because the dash is a minus. `empty` is true for `null`, `""` and empty collections. Book: Section 7.13.3.*

JSTL: the Last of the Java Comes Out

Table 7.7 The core JSTL tags. The other libraries are `fmt` for numbers, dates and messages, `fn` for string functions such as `fn:length` and `fn:substring`, and `sql`, which you should not use.

Tag	What it does
`<c:out value="${x}"/>`	Prints `x`, **escaping** the HTML special characters
`<c:out value="${x}" default="-"/>`	Prints `-` when `x` is empty
`<c:set var="n" value="${x}" scope="session"/>`	Puts a value into a scope
`<c:remove var="n" scope="session"/>`	Takes it out again
`<c:if test="${x gt 3}"> … </c:if>`	Includes its body when the test is true
`<c:choose><c:when test="…">…</c:when><c:otherwise>…</c:otherwise></c:choose>`	The `if` / `else if` / `else` chain
`<c:forEach var="b" items="${bills}"> … </c:forEach>`	Repeats its body once per item
`<c:forEach … varStatus="st">`	Gives `st.index`, `st.count`, `st.first`, `st.last`
`<c:url value="/bills"/>`	Builds a link with the context path
`<c:redirect url="/bills"/>`	Sends a redirect

***JSTL** (Jakarta Tags, **Table 7.7**): EL removed the Java that reads; JSTL removes the Java that decides and repeats. `<c:forEach>` becomes a plain loop, so it is not slower. Book: Sections 7.14.1, 7.14.2 and 7.14.4.*

`<c:out>` and the Attack It Stops

Listing 7.15 The same value printed two ways. The first is a hole; the second is not.

```
Customer: <%= request.getParameter("name") %>
Customer: <c:out value="${param.name}"/>
```

> **Bare `${x}` in template text does not escape.** Fine for a bill id or a formatted amount; a hole for anything a person typed: a name, a phone, a filter box.

*With the name `<script>alert('x')</script>`, the first line of **Listing 7.15** runs the script in every browser; the second draws it as text. Validate on input, escape on output. Book: Section 7.14.3.*

Model–View–Controller

DEFINITION. The **model** is the data and its rules, knowing nothing of the web; the **view** shows; the **controller** receives the request, asks the model, and chooses the view. Here: a servlet, a JSP page, and plain Java classes.

THE KEY INSIGHT. The controller contains no HTML and the view contains no decisions. Nearly every later difficulty comes from crossing one of those two lines.

The servlet's strength is controlling and dispatching; the page's is displaying. Book: Section 7.15.1.

The Laundry Pages as MVC

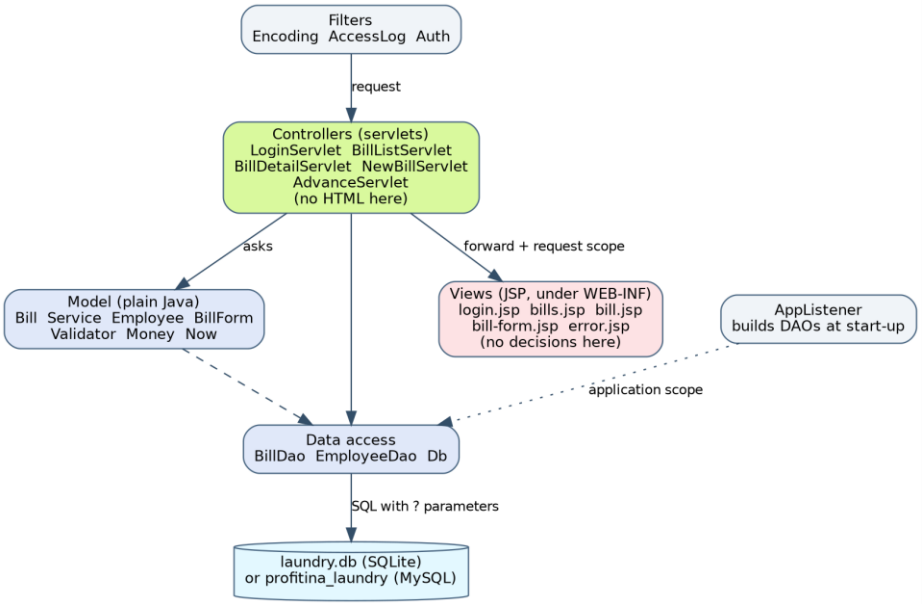


Figure 7.3 The Laundry bills pages as Model–View–Controller. Controllers write no HTML; views make no decisions; the model knows nothing of HTTP. The same model and DAO classes are reused unchanged by the JDK HttpServer version of Section 7.21.

***Figure 7.3**: five servlets, five views under `WEB-INF/views`, a model of plain classes with `BillDao` and `EmployeeDao`, three filters in front, and a listener that builds the DAOs at start-up. Book: Section 7.15.2.*

The Controller for `GET /bills`

Listing 7.16 `doGet` of `BillListServlet`: read two filters, ask the model, put the result in request scope, forward. No HTML.

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String outlet = Validator.field(req.getParameter("outlet"));
        String status = Validator.field(req.getParameter("status"));
        if (!status.matches("queued|finished|departed")) {
            status = "";                                  // anything else means "all"
        }
        BillDao dao = (BillDao) getServletContext().getAttribute("billDao");
        try {
            req.setAttribute("bills", dao.find(outlet, status, 50));
        } catch (SQLException e) {
            throw new ServletException("could not read the bills", e);
        }
        req.setAttribute("outlet", outlet);
        req.setAttribute("status", status);
        req.getRequestDispatcher("/WEB-INF/views/bills.jsp").forward(req, resp);
    }
```

*In **Listing 7.16** a missing parameter is `""`, an unknown status means "all", the DAO comes from application scope, and the view is reachable only by this forward. Book: Section 7.15.3.*

The View: EL and JSTL Only

Listing 7.17 The loop of `bills.jsp`: names and descriptions go through `<c:out>`; ids, amounts and status words are the application's own.

```
      <c:forEach var="b" items="${bills}" varStatus="st">
        <tr class="${b.status}">
          <td>${st.count}</td>
          <td><a href="${ctx}/bill?id=${b.id}">${b.id}</a></td>
          <td>${fn:substring(b.arrived, 0, 16)}</td>
          <td><c:out value="${b.customerName}"/></td>
          <td><c:out value="${b.serviceDescription}"/></td>
          <td>${b.quantity}</td>
          <td class="num">${b.paymentRupiah}</td>
          <td>${b.cancelled ? 'cancelled' : b.status}</td>
        </tr>
      </c:forEach>
```

Listing 7.17 *begins by including* `header.jspf`*, which sets* `ctx`*, declares the tag libraries, shows who is logged in and prints a one-time flash message. Book: Section 7.15.3.*

The Bill List, as the Cashier Sees It

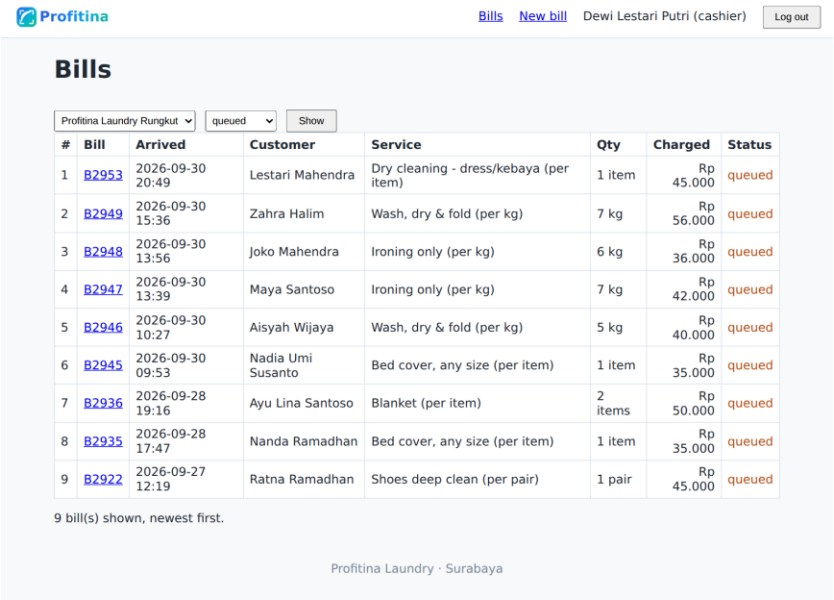


Figure 7.4 The bill list filtered to outlet O01 and status *queued*: nine bills, newest first, against the sample database. Screenshot of the HttpServer track of Section 7.21, which renders the same markup as `bills.jsp`.

Figure 7.4 *comes from the JDK HttpServer track, which renders the same markup as* `bills.jsp` *and was run for real. Book: Section 7.15.3.*

The Form: Validate, Redisplay, or Store and Redirect

Listing 7.18 From `doPost` of `NewBillServlet`: a bad form comes back with status 400; a good one is stored at a price computed here, and the browser is redirected.

```
        BillDao dao = (BillDao) getServletContext().getAttribute("billDao");
        try {
            Map<String, String> errors = Validator.validate(form, dao);
            if (!errors.isEmpty()) {
                resp.setStatus(HttpServletResponse.SC_BAD_REQUEST);
                show(req, resp, form, errors);
                return;
            }
            Service service = dao.service(form.getServiceId());
            int units = Validator.units(service, form.getQuantity());
            long price = service.getPrice() * units;       // BR-1: computed here, never sent by the browser
            String id = dao.insert(form.getCustomerId(), service.getId(), units, price,
                                   form.getRack(), form.getOutletId(), user.getId(), Now.text());
            req.getSession().setAttribute("flash", "Bill " + id + " created: " + units + " "
                    + service.getUnitWord() + " x " + service.getPriceRupiah() + " = " + Money.rupiah(price) +
"." );
            resp.sendRedirect(req.getContextPath() + "/bill?id=" + id);
```

*Before **Listing 7.18** the role is checked on the server and the form copied field by field. **Post/Redirect/Get:** Reload repeats the harmless GET of `/bill?id=B2954`, never the POST. The flash message survives the redirect in session scope. Book: Section 7.15.4.*

Worked Example 4: a New Bill

Listing 7.19 `Validator.units`: kilograms are rounded up to a whole number with `BigDecimal`; items must already be whole.

```
    public static int units(Service service, String quantity) {
        try {
            BigDecimal q = new BigDecimal(quantity.replace(',', '.'));
            if (q.signum() <= 0 || q.compareTo(BigDecimal.valueOf(100)) > 0) {
                return -1;
            }
            if (service.isPerKg()) {
                return q.setScale(0, RoundingMode.CEILING).intValueExact();
            }
            return q.stripTrailingZeros().scale() <= 0 ? q.intValueExact() : -1;
        } catch (NumberFormatException | ArithmeticException e) {
            return -1;
        }
    }
```

Customer `c0001`, S0002 (express, Rp 12.000 per kg), weight `3.2`, rack `a02`: ids upper-cased to `C0001`, `A02`; 3.2 kg rounds **up** to 4 units; 4 × 12000 = 48000.

*The highest id is B2953, so the new bill is B2954, stored `queued` with its `StatusLog` row: "Bill B2954 created: 4 kg x Rp 12.000 = Rp 48.000." (**Listing 7.19**). All three tracks printed exactly this. Book: Section 7.15.4.*

JDBC Through a DAO

> **DEFINITION.** A **Data Access Object** holds every statement touching storage and exposes methods in the application's language: `find`, `findById`, `insert`, `advance`.

Listing 7.20 The start of `BillDao.find`: one fixed SQL text, five placeholders bound one by one.

```
    public List<Bill> find(String outletId, String status, int limit) throws SQLException {
        String sql = SELECT
            + " WHERE (? = '' OR b.Outlet_id = ?) AND (? = '' OR b.status = ?)"
            + " ORDER BY b.arrived DESC, b.id DESC LIMIT ?";
        try (Connection c = db.open(); PreparedStatement ps = c.prepareStatement(sql)) {
            ps.setString(1, outletId);
            ps.setString(2, outletId);
            ps.setString(3, status);
            ps.setString(4, status);
            ps.setInt(5, limit);
```

*In **Listing 7.20** `(? = '' OR b.Outlet_id = ?)` means "any outlet when the filter is empty"; an "outlet" of `' OR '1'='1` matches nothing. `insert` wraps the bill and its log row in `setAutoCommit(false)`, `commit()`, `rollback()`. The JDBC URL, or a JNDI pool, picks SQLite or MySQL. Book: Section 7.16.*

Logging In

Listing 7.21 The end of `LoginServlet.doPost`: one answer for both mistakes; on success a new session id, the employee and a CSRF token, and a redirect.

```
        if (user == null) {                                  // one answer for both mistakes
            req.setAttribute("error", "E-mail or password is not correct.");
            req.setAttribute("email", email);
            resp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            req.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(req, resp);
            return;
        }
        HttpSession session = req.getSession();
        req.changeSessionId();                               // a new id at login: no session fixation
        byte[] token = new byte[16];
        RANDOM.nextBytes(token);
        session.setAttribute("user", user);
        session.setAttribute("csrf", HexFormat.of().formatHex(token));
        resp.sendRedirect(req.getContextPath() + "/bills");
```

***Listing 7.21**: one message so nobody learns which e-mails exist; passwords checked against a PBKDF2 hash in constant time; `changeSessionId()` defeats session fixation; the CSRF token lets the filter refuse forged POSTs. Never print a session id. Book: Section 7.17.*

Filters

> **DEFINITION.** A **filter** runs before the servlet for every matching address; `chain.doFilter(request, response)` passes the request on, and a filter that does not call it stops the request there.

Listing 7.22 The body of `AuthFilter.doFilter`: no employee in the session, go to login; a POST without the right token, 403; otherwise pass on.

```
        HttpSession session = req.getSession(false);          // do not create one
        if (session == null || session.getAttribute("user") == null) {
            resp.sendRedirect(req.getContextPath() + "/login");
            return;                                           // the chain stops here
        }
        if (req.getMethod().equals("POST")) {
            Object expected = session.getAttribute("csrf");
            if (expected == null || !expected.equals(req.getParameter("csrf"))) {
                resp.sendError(HttpServletResponse.SC_FORBIDDEN, "Form expired; please reload the page.");
                return;
            }
        }
        chain.doFilter(req, resp);
```

*Thanks to **Listing 7.22** no bill servlet contains a login check, and none can forget one. Filters hold cross-cutting rules: middleware in ASP.NET Core, `before_request` in Flask. Book: Section 7.18.*

Errors, Includes, Forwards

Table 7.8 The two include mechanisms. The last four rows decide which to use.

	`<%@ include %>`	`<jsp:include>`
When	Translation time	Request time
What is inserted	The other file's **text**	The other page's **output**
One servlet or two	One	Two
Can the file name be computed?	No	Yes
Sees the including page's variables	Yes — same method	No — separate page
Cost per request	None	A method call

> **THE KEY INSIGHT: a forward is not a redirect.** A forward stays inside the server: one request, the same address bar, request scope intact. A redirect sends 302 or 303 and the browser asks again. Forward to show a view; redirect after a successful POST.

*From **Table 7.8**: the directive for content that changes rarely, `<jsp:include>` for content chosen per request. `error.jsp` shows one sentence and logs the exception: never show a stack trace. Book: Section 7.19.*

One Application, Three Stacks

Table 7.9 Part of the translation table between the three stacks. Only the names change; the jobs are the same.

Idea	Servlet + JSP (Tomcat)	JDK HttpServer	Flask + Jinja2
Address → code	`@WebServlet("/bills")`	`createContext("/bills", h)`	`@app.route("/bills")`
Code before every request	`Filter` / `HttpFilter`	`com.sun.net.httpserver.Filter`	`@app.before_request`
Session	`HttpSession` (server side)	`Sessions` (written by hand)	`flask.session` (signed cookie)
Print, escaped	`<c:out value="${x}"/>`	`{{x}}`	`{{ x }}`
Loop	`<c:forEach>`	Java loop over a row template	`{% for b in bills %}`
Forward to a view	`RequestDispatcher.forward`	`Template.render`	`return render_template(…)`

***Table 7.9**: stand-in container objects (16 checks), the JDK server over real HTTP (14), and Flask's test client (11) all printed "Bill B2954 created: 4 kg x Rp 12.000 = Rp 48.000." Jinja2 escapes by default. Book: Sections 7.20–7.22.*

JSP Next to PHP

Table 7.10 PHP and servlet/JSP side by side. The last two rows are the only differences of kind.

Idea	PHP (Chapters 3 and 5)	Servlet + JSP
A hole in a page	`<?php … ?>`	`<% … %>`
Print a value	`<?= $x ?>`	`<%= x %>` or `${x}`
Query or form value	`$_GET['q']`, `$_POST['name']`	`request.getParameter("q")`, `${param.q}`
Escape for HTML	`htmlspecialchars($x)`	`<c:out value="${x}"/>`
Session	`$_SESSION['k']` after `session_start()`	`session.setAttribute("k", v)`
Prepared statement	PDO with `?` placeholders	`PreparedStatement` with `?`
When your code runs	interpreted, every request	translated and compiled once
Between requests	nothing survives	the servlet, the page object and their fields survive

*The last two rows of **Table 7.10** are what a PHP developer must unlearn: a Java field is shared by every visitor, and a URL maps to a component, not a file. Book: Section 7.23.*

Profitina in Practice

Profitina (`app.profitina.com`) is Python on FastAPI and PostgreSQL, with no Java, and every idea of today carries weight in it.

Controllers without HTML, views without decisions: the dashboard shows a score and computes nothing.

Filters: authentication, a login rate limit and consent checks run before any handler. **Escaping without exception** in the community feed.

Scopes and secrets: a scan per request, the user per session; Argon2id hashes; UU PDP duties as features.

The runnable case study is always Profitina Laundry; the shape carries across languages. Book: Section 7.24.

Common Mistakes

Setup: `javax` imports on Tomcat 10 or 11; the Servlet API jar inside the WAR; no JSTL jars; `admin`/`admin`.

Pages: a semicolon inside `<%= … %>`; request data in a declaration or servlet field; bare `${x}` for typed values; `${prices.wash-fold}`; `<a href="/bills">`.

Requests: forgetting that `getParameter` returns `null`; setting the encoding after the first `getParameter`; `property="*"` on a bean with a price.

Design: forwarding after a successful POST; SQL built by joining strings; a stack trace shown to a visitor; `HttpExchange.setAttribute` trusted as per-request.

Each mistake is listed in the book with its fix. Book: Section 7.25.

Summary: Five Things to Remember

1. A container runs your classes: connector, filters, servlet, JSP page. Use Tomcat 10.1 or 11 and `jakarta.*`.

2. A JSP page is translated into a servlet once; every request is one call to `_jspService` on one shared object.

3. Nine implicit objects and four scopes; a declaration is shared by everyone.

4. Beans, EL and JSTL take the Java out of the page; `<c:out>` escapes, and that default is a security control.

5. Servlet and JSP together are MVC; a DAO binds every value; filters guard; redirect after a successful POST.

The same application ran on Tomcat's API, on the JDK's own server and in Flask: only the names change. Book: Section 7.26.

Next Time

Chapter 8 — Evaluation and Discussion II. Chapters 5 to 7 worked in full: a CRUD website with authentication, and new problems on PHP and MySQL, XML and JSON, servlets and JSP.

Every problem is solved in PHP, Java and Python, and checked by a program.

Before then: work Exercises 7.2, 7.5, 7.6, 7.8 and 7.9, and reread Chapters 5 and 6.

Solutions to every exercise are in the book. Book: Sections 7.27 and 7.28.

Lecture 8 — Evaluation and Discussion II: A Database-Backed Web Application, from Design to Deployment

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 8

Lecturer: Irfan Subakti — yifana@gmail.com

No new technology today: twelve problems, each solved from a blank page in PHP, SQL, XML, JSON or JSP, and again in Java and Python, and every output printed by the program. Book: Chapter 8, Sections 8.1–8.11.

What This Meeting Covers

1. **A method** — eight steps from a paragraph of requirements to a tested web application.
2. **The project, Problems 8.1–8.4** — the database design, registration and login, eight features, and the evidence and delivery.
3. **PHP and MySQL, Problems 8.5–8.7** — a report with `GROUP BY` and `HAVING`, a check that injection cannot trick, a cookie that is not a session.
4. **XML and JSON, Problems 8.8–8.10** — an export that stays well-formed, a schema that catches nonsense, prices that are numbers.
5. **Servlets and JSP, Problems 8.11–8.12** — model, view and controller; four places to keep a number.

Each problem goes the same route: the problem, in our own words, design, the solutions, line by line, a tempting wrong solution, and a check-yourself variant. Book: Section 8.1.

Where We Stand: the Toolbox

Chapter 5: a running server, redirection, cookies and sessions, the relational model and normalization, SQL from `CREATE TABLE` to `HAVING`, `mysqli` and PDO, prepared statements, transactions.

Chapter 6: well-formedness, entities, the tree, namespaces, well-formed against valid, DTD and XML Schema, DOM, SAX, XPath, XSLT, and JSON.

Chapter 7: the container, servlets and their lifecycle, JSP, implicit objects, the four scopes, beans, EL, JSTL, forwarding, and JDBC through a DAO.

> **HOW TO USE TODAY.** Read a statement, cover the rest, and spend ten minutes on it: sketch the tables, write the URL of every page, write one SQL statement by hand. Then compare.

A solution you have wrestled with first is understood in a way that a solution you have only read never is. Book: Section 8.1.

The Method, Steps 1 to 4

> **DEFINITION.** A *requirement* is one sentence that says what the system must let someone do, or must never let happen. A *feature* is a group of pages and actions that satisfies one or more requirements for one kind of user.

1. People and verbs: customer, cashier, visitor; *register, log in, search, add, pay, collect.*

2. Nouns become tables; "has" becomes a foreign key on the "many" side. **3. Every fact in one place**, except a copy that records history, such as the amount charged.

4. List the URLs: reading with `GET`, changing with `POST`, and a redirect after every successful `POST` (Post/Redirect/Get).

Beginners start by writing `<html>`: the one place not to start, because every page depends on data that has not been decided yet. Book: Section 8.2.

The Method, Steps 5 to 8

5. Who may do what, for every URL, checked on the server for every request. Hiding a link is a convenience, not a protection.

6. Write the checks before the code, with an edge case each: the empty search, the quantity zero, the customer who still has bills.

7. One request at a time, run after each route. **8. Secrets out of the code,** and a short write-up kept as you go.

> **THE ORDER MATTERS.** Data, then addresses, then permissions, then tests, and only then code. A team that starts at Step 7 redesigns its tables on the last evening.

On a small problem the first four steps take twenty minutes; on a large one they are the difference between a design and a heap of pages. Book: Section 8.2.

One Database, One Folder of Code

Table 8.1 The main folders of the chapter's code. Each program names, in its first comment, the command that runs it.

Folder	What is in it	Needs
`data/`	`laundry.db`, a copy of the Profitina Laundry database	nothing
`sql/`	the SQL of Problems 8.1 and 8.5 and of the exercises	SQLite or MySQL
`php/`	the Laundry Desk (`lib.php`, `public/index.php`) and Problems 8.2 and 8.5 to 8.10	PHP 8 with PDO
`java/`	the Laundry Desk and every problem on the JDK's own HTTP server and JDBC	JDK 21 or newer
`python/`	the Laundry Desk and every problem on Flask and `sqlite3`	Python 3.11+, Flask, lxml
`jsp/`	the servlets, the JSP page and a small stand-in container (Problems 8.11, 8.12)	JDK, servlet API jar
`tests/`	the acceptance test of Problem 8.4, and `run_all.sh`, which runs everything	all of the above

The Profitina Laundry database: three outlets, fourteen employees, twelve services, 256 customers, 2,953 bills, 3,319 payments, one year ending 30 September 2026 at 21:00. SQLite by default; MySQL with one environment variable (Table 8.1). Book: Section 8.3.

The Project: a CRUD Website with Authentication

> **PROBLEM.** Build a CRUD website with Laravel or another framework: at least eight features beyond authentication; login, registration and logout; a schema with at least one foreign key. Write a report with the conceptual and physical models, the features with screenshots, and each member's share; deploy it to a free host and publish the source.

In our own words: four problems, solved in the order they depend on each other. **8.1** the database; **8.2** registration, login, logout; **8.3** eight features; **8.4** evidence and delivery.

*The **Laundry Desk** is built three times without a framework (PHP with PDO, Java on the JDK's HTTP server, Python with Flask), each in the MVC way, so that moving to Laravel afterwards is a translation, not a mystery. Book: Section 8.4.*

Problem 8.1 — The Conceptual Model

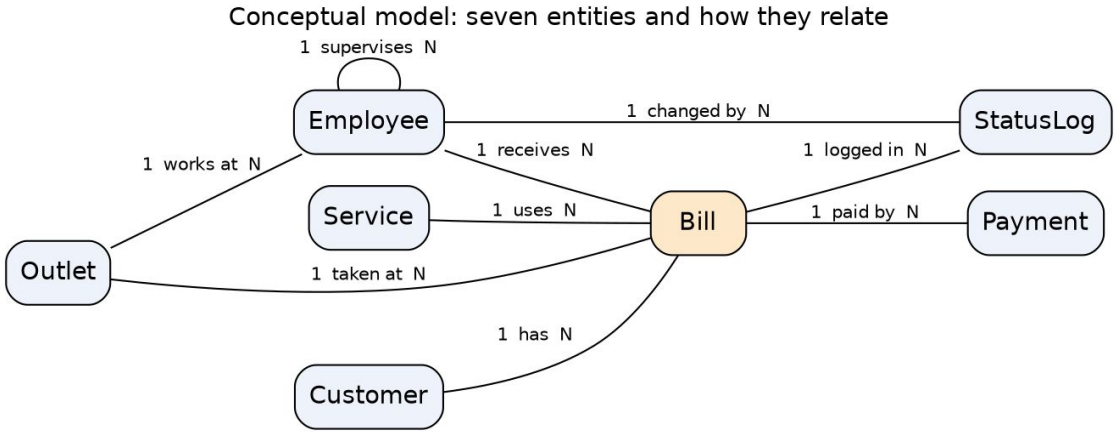


Figure 8.1 The conceptual model of Profitina Laundry: seven entities, with the bill at the center. Every line is one-to-many, read from the "1" side to the "N" side.

A conceptual model lists entities and relationships with their cardinality. Read each line of Figure 8.1 both ways: "one customer has many bills; one bill belongs to exactly one customer". The owner, who has no supervisor, is why the supervisor column may be empty. Book: Section 8.4.1.

Problem 8.1 — The Physical Model

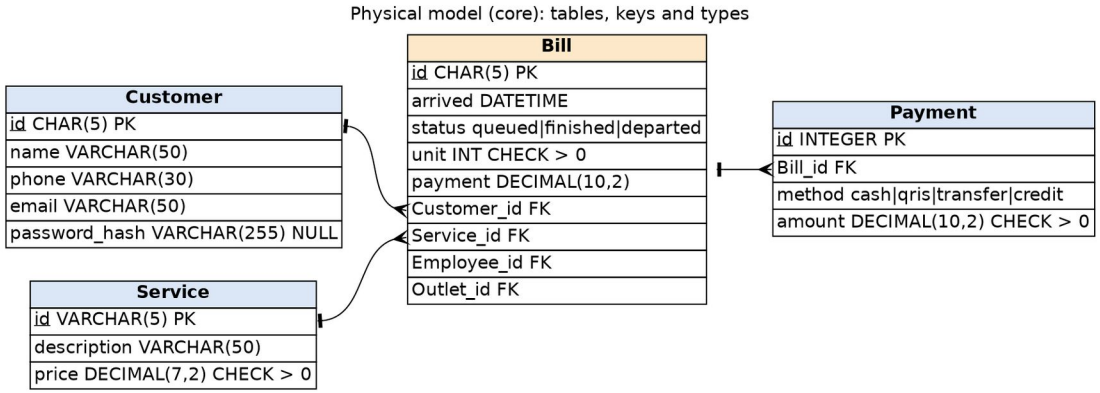


Figure 8.2 The core of the physical model: primary keys underlined, each foreign key drawn from the key it refers to. The crow's foot marks the "many" end.

Figure 8.2: readable codes (`C0001`, `B2953`) because they are printed on receipts; money as `DECIMAL(10,2)`; `Bill.payment` is the amount charged, copied at creation; `password_hash` may be `NULL` for customers with no account. Book: Section 8.4.1.

Problem 8.1 — The `Bill` Table in MySQL

Listing 8.1 The `Bill` table of `p81_design.mysql.sql`: named constraints, a `CHECK`, and two foreign keys with `RESTRICT` and `CASCADE`.

```
CREATE TABLE Bill (
    id          CHAR(5)       NOT NULL,
    arrived     DATETIME      NOT NULL,
    unit        INT           NOT NULL,
    payment     DECIMAL(10,2) NOT NULL,
    Customer_id CHAR(5)       NOT NULL,
    Service_id  VARCHAR(5)    NOT NULL,
    CONSTRAINT Bill_pk PRIMARY KEY (id),
    CONSTRAINT Bill_unit_pos CHECK (unit > 0),
    CONSTRAINT Bill_Customer FOREIGN KEY (Customer_id) REFERENCES Customer (id)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT Bill_Service FOREIGN KEY (Service_id) REFERENCES Service (id)
        ON DELETE RESTRICT ON UPDATE CASCADE
) ENGINE = InnoDB;
```

Constraints in Listing 8.1 are named, so that an error can say which rule was broken. `ENGINE = InnoDB` enforces foreign keys; the old MyISAM engine accepts the clause and silently ignores it. Book: Section 8.4.1.

Problem 8.1 — Putting the Keys to Work

Listing 8.2 The four test statements of `p81_design.mysql.sql`.

```sql
-- 1. A bill for a customer who does not exist: the foreign key refuses it.
INSERT INTO Bill VALUES ('B0002', '2026-09-30 08:20:00', 2, 40000, 'C9999', 'S0004');
-- 2. Deleting a customer who still has a bill: RESTRICT refuses it.
DELETE FROM Customer WHERE id = 'C0001';
-- 3. Deleting a customer with no bills is allowed.
DELETE FROM Customer WHERE id = 'C0002';
-- 4. Changing a customer's key: CASCADE carries the change into Bill.
UPDATE Customer SET id = 'C0100' WHERE id = 'C0001';
```

```
ERROR 1452 (23000) at line 45: Cannot add or update a child row: a foreign key constraint fails
(`p81_demo`.`Bill`, CONSTRAINT `Bill_Customer` FOREIGN KEY (`Customer_id`) REFERENCES `Customer` (`id`)
ON UPDATE CASCADE)
--------------
DELETE FROM Customer WHERE id = 'C0001'
--------------

ERROR 1451 (23000) at line 47: Cannot delete or update a parent row: a foreign key constraint fails
(`p81_demo`.`Bill`, CONSTRAINT `Bill_Customer` FOREIGN KEY (`Customer_id`) REFERENCES `Customer` (`id`)
ON UPDATE CASCADE)
```

In Listing 8.2 statement 1 fails with 1452, a child row pointing at nothing; statement 2 fails with 1451, `C0001` still has `B0001`; statement 3 succeeds; statement 4 carries `C0100` into `Bill` by `ON UPDATE CASCADE`. Run with `mysql -u root --force -t`. Book: Section 8.4.1.

Problem 8.1 — A Tempting Wrong Solution

Leave out `PRAGMA foreign_keys = ON` on SQLite, and every statement "succeeds":

```
INSERT INTO Customer VALUES ('C0001', 'Nabila Rahman', '0896 OK, 2 row(s)
INSERT INTO Service VALUES ('S0001', 'Wash, dry & fold (per  OK, 2 row(s)
INSERT INTO Bill VALUES ('B0001', '2026-09-30 08:15:00', 5,  OK, 1 row(s)
INSERT INTO Bill VALUES ('B0002', '2026-09-30 08:20:00', 2,  OK, 1 row(s)
DELETE FROM Customer WHERE id = 'C0001'                      OK, 1 row(s)
DELETE FROM Customer WHERE id = 'C0002'                      OK, 1 row(s)
UPDATE Customer SET id = 'C0100' WHERE id = 'C0001'          OK, 0 row(s)
SELECT ->
SELECT ->
    B0001 | C0001
    B0002 | C9999
```

The bill for `C9999` is stored, `C0001` is deleted under his bill, and two orphan bills remain. SQLite checks foreign keys only when each connection asks, so every program switches them on at its second line of database code. Book: Section 8.4.1.

Problem 8.1 — Check Yourself

> **CHECK YOURSELF.** A table `Pickup(id, Bill_id, courier, pickup_at)` records couriers collecting laundry; every pickup belongs to exactly one bill, and a bill is picked up at most once. Which constraint says "at most once"?

The foreign key `Pickup_Bill FOREIGN KEY (Bill_id) REFERENCES Bill (id)` says "a real bill".

`UNIQUE (Bill_id)` says "at most once": the relationship becomes one-to-one.

The full `CREATE TABLE`, with `ON DELETE RESTRICT ON UPDATE CASCADE` and `ENGINE = InnoDB`, is in the book. Book: Section 8.4.1.

Problem 8.2 — Registration, Login, Logout

> **PROBLEM.** The website must include login, registration and logout features.

Registration is for customers: a new `Customer` row with a password hash. **Login** looks in `Employee`, then `Customer`, and remembers *staff* or *customer*.

Rule 1. Never store a password: store a slow, salted hash, PBKDF2 with HMAC-SHA256 and 600,000 rounds.

Rule 2. Keep the identity on the server. **Rule 3.** A new session id at login (against *session fixation*); destroy the session at logout.

The stored form is `pbkdf2_sha256$600000$<salt>$<hash>`, 90 characters. Book: Section 8.4.2.

Problem 8.2 — Hashing and Checking

Listing 8.3 `hash_password` in `lib.php`: a fresh 16-byte salt and 600,000 rounds of PBKDF2.

```
function hash_password(string $password): string
{
    $salt = random_bytes(16);
    $key = hash_pbkdf2('sha256', $password, $salt, ITERATIONS, 32, true);
    return 'pbkdf2_sha256$' . ITERATIONS . '$' . base64_encode($salt) . '$' . base64_encode($key);
}
```

Listing 8.4 The heart of `verify_password` in `lib.php`: recompute with the stored salt and rounds, then compare in constant time.

```
    $salt = base64_decode($p[2], true);
    $expected = base64_decode($p[3], true);
    if ($salt === false || $expected === false) {
        return false;
    }
    $key = hash_pbkdf2('sha256', $password, $salt, (int) $p[1], strlen($expected), true);
    return hash_equals($expected, $key);            // constant-time comparison
```

In Listing 8.3 the salt is new for every password; Listing 8.4 first refuses any stored form it does not recognize, then compares with `hash_equals`. Java uses `SecretKeyFactory` and `MessageDigest.isEqual`; Python `hashlib.pbkdf2_hmac` and `hmac.compare_digest`. Book: Section 8.4.2.

Problem 8.2 — The Login Route

Listing 8.5 The POST branch of `/login` in `public/index.php`: look in `Employee`, then in `Customer`; on success a new session id, then the user.

```
    if ($method === 'POST') {
        $email = $in('email');
        $u = row('SELECT id, name, Outlet_id, password_hash FROM Employee WHERE email = ?', [$email]);
        $kind = 'staff';
        if ($u === null) {
            $u = row('SELECT id, name, NULL AS Outlet_id, password_hash FROM Customer WHERE email = ?',
[$email]);
            $kind = 'customer';
        }
        if ($u !== null && verify_password($_POST['password'] ?? '', $u['password_hash'])) {
            session_regenerate_id(true);              // a fresh session id after every login
            $_SESSION['user'] = ['kind' => $kind, 'id' => $u['id'], 'name' => $u['name'],
                                 'outlet' => $u['Outlet_id']];
            flash('Welcome, ' . $u['name'] . '.');
            redirect('/home');
        }
```

In Listing 8.5 the message is the same for an unknown address and a wrong password, so the form cannot reveal who has an account. The session was started with `cookie_httponly` and `cookie_samesite => 'Lax'`. Book: Section 8.4.2.

Problem 8.2 — A Tempting Wrong Solution

"The password only needs to be scrambled": `hash('sha256', $password)`, no salt, no rounds. Two customers choose the same password:

```
unsalted sha256, customer A: bee5688aea66a474...
unsalted sha256, customer B: bee5688aea66a474...
identical?                   yes
PBKDF2 stored forms equal?   no
stored form length:          90 characters
A verifies with rahasia123?  yes
A verifies with rahasia124?  no
```

The unsalted hashes are identical, and SHA-256 is fast: billions of guesses per second on a graphics card. With PBKDF2 and a fresh salt the stored forms differ and every guess costs 600,000 rounds. Java and Python printed the same seven lines. Book: Section 8.4.2.

Problem 8.3 — Nine Features

Table 8.2 The nine features of the Laundry Desk, beyond registration, login and logout. Every letter of CRUD appears, and every feature is a URL a test can call.

#	Feature	Who	URLs	CRUD
F1	List and search customers	staff	`GET /customers?q=`	R
F2	Add a customer	staff	`GET`, `POST /customers/new`	C
F3	Edit a customer	staff	`GET`, `POST /customers/{id}/edit`	U
F4	Delete a customer (refused while bills exist)	staff	`POST /customers/{id}/delete`	D
F5	The price list	everyone	`GET /services`	R
F6	Take in a new bill, priced by the server	staff	`GET`, `POST /bills/new`	C
F7	Move a bill to its next status, with an audit row	staff	`POST /bills/{id}/advance`	U, C
F8	Show a bill, record a payment, keep the balance	staff	`GET /bills/{id}`, `POST /bills/{id}/pay`	R, C
F9	A customer's own bills and balances	customer	`GET /my-bills`	R

Every letter of CRUD appears in Table 8.2, and every feature is a URL a test can call. F9 gives the registration of Problem 8.2 something to be for. Book: Section 8.4.3.

One Request Through the Laundry Desk

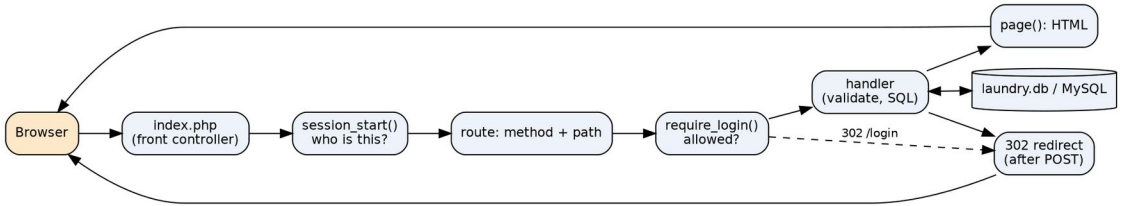


Figure 8.3 One request through the Laundry Desk. A failed permission check short-cuts to a redirect; a successful change ends in a redirect too, so that reloading the page never repeats the change.

Figure 8.3: session, route, permission check, validation, database; a page for a `GET`, a redirect after a `POST` that changed something. Book: Section 8.4.3.

The Life of a Bill

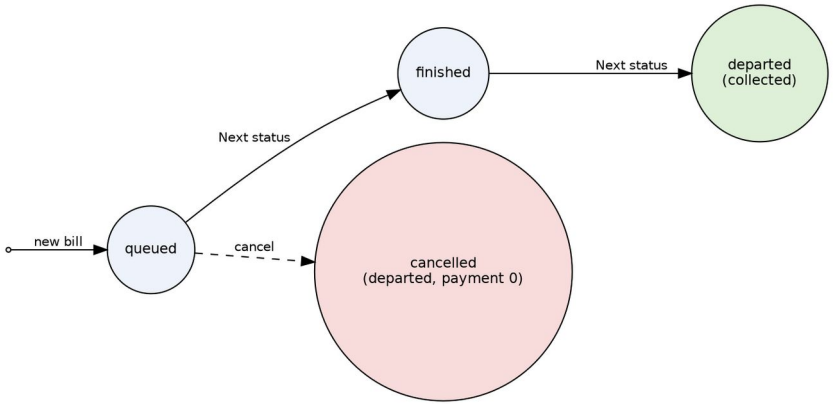


Figure 8.4 The life of a bill. The button *Next status* (feature F7) follows the solid arrows only; the dashed arrow is the cancellation of Exercise 8.10.

Figure 8.4: `queued`, `finished`, `departed`; *the status never moves backwards, and every move adds a* `StatusLog` *row naming the employee. Book: Section 8.4.3.*

F6: a New Bill, Priced by the Server

Listing 8.6 The storing branch of `/bills/new` in `public/index.php`: one transaction, the next id, the price copied into the bill, and the first log row.

```
        } else {
            $pdo = db();
            $pdo->beginTransaction();                // the bill and its first log row, or neither
            $id = next_id('Bill', 'B');
            $total = (int) $svc['price'] * $unit;    // the price is copied into the bill now
            run("INSERT INTO Bill (id, arrived, status, rack, unit, payment, Customer_id, Employee_id,
                 Service_id, Outlet_id) VALUES (?,?, 'queued', ?,?,?,?,?,?,?)",
                [$id, now(), $rack, $unit, $total, $cust['id'], $u['id'], $svc['id'], $u['outlet']]);
            run("INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) VALUES (?,
'queued', ?, ?)",
                [$id, now(), $u['id']]);
            $pdo->commit();
            flash("Bill $id created: " . rupiah($total) . '.');
            redirect("/bills/$id");
```

Before Listing 8.6 the handler checks the customer, the service, `unit` from 1 to 100 and the rack. The price is never a form field: a charge taken from the browser is a charge the customer can choose. Book: Section 8.4.3.

F4: the Database Says No

Listing 8.7 Feature F4 in `public/index.php`: try the `DELETE`; the foreign key `Bill_Customer` refuses it while bills exist.

```
if ($method === 'POST' && preg_match('#^/customers/(C\d{4})/delete$#', $path, $m)) {
    try {
        run('DELETE FROM Customer WHERE id = ?', [$m[1]]);
        flash("Customer {$m[1]} deleted.");
    } catch (PDOException $e) {                    // the foreign key Bill_Customer said no
        flash("Customer {$m[1]} has bills and cannot be deleted.");
    }
    redirect('/customers');
}
```

Listing 8.7 lets the database enforce the rule and reports it in words. F8 refuses a payment larger than the balance, computed as the charge minus `COALESCE(SUM(p.amount), 0)`*. Book: Section 8.4.3.*

Where the MVC Is, and Where Laravel Puts It

Table 8.3 Each piece of the PHP solution and its Laravel counterpart. The ideas are the same; Laravel supplies them ready-made.

In the Laundry Desk (PHP)	In Laravel
`public/index.php`, the list of `if ($path === …)`	`routes/web.php`, `Route::get('/customers', …)`
the handler code of one feature	a method of a controller class, `CustomerController@index`
`rows()`, `row()`, `run()` with `?` placeholders	Eloquent models (`Customer::where(…)->get()`) or the query builder
`p81_design.mysql.sql`, `CREATE TABLE …`	a migration, `Schema::create('Customer', …)`
`page()` and string-built HTML with `h()`	Blade templates, where `{{ $name }}` escapes automatically
`require_login('staff')`	the `auth` middleware plus a policy or gate
`hash_password()`, `verify_password()`	`Hash::make()`, `Hash::check()` (bcrypt or Argon2 by default)

The model is the database with its constraints plus the SQL; the view turns data into HTML; the controller routes and decides (Table 8.3). The Laravel commands are in the book without output: they need downloads the build machine could not make. Book: Section 8.4.3.

Problem 8.3 — Trusting the Browser

The tempting solution: `<input name="unit" type="number" min="1" max="100" required>` refuses 0, so the server need not check.

Why it is wrong: anybody can send the request without the form, with `curl` or Python; the acceptance check F6 sends 0 and all three servers answer "The quantity must be a whole number from 1 to 100."

> **CHECK YOURSELF.** A tenth feature, "complaints": which table, which status, which checks? **Answer:** a `StatusLog` row with `status = 'complaint'`; the server checks that the user is staff, the bill exists, and its status is `departed`.

The browser's check is a courtesy to honest users, not a protection. Book: Section 8.4.3.

Problem 8.4 — Evidence That Can Be Rerun

> **PROBLEM.** Show that each feature works, write the report, deploy the website to a free host, and publish the source in a public repository.

An acceptance test uses the site exactly as a browser does: it sends forms, keeps cookies, never follows a redirect by itself, and reads only fixed marks (`<h1>`, `<p class="msg">`, `<p class="err">`, `<tr><td>`).

Because it knows nothing else, the same `acceptance.py` checks the PHP, the Java and the Python server; `run_acceptance.sh` gives it a fresh database each time.

Screenshots prove that a feature worked once; a test re-proves it in two seconds after every change. Book: Section 8.4.4.

Worked Example: the Acceptance Run, Authentication

```
A1  GET /customers, not logged in              -> 302 /login
A2  POST /register Rina Wati                   -> 302, Account C0257 created. Please log in.
A3  POST /register the same e-mail again       -> That e-mail address already has an account.
A4  POST /login with a wrong password          -> Wrong e-mail or password.
A5  POST /login Rina, right password           -> 302 /home
A6  GET /my-bills as Rina                      -> 200, 'Welcome, Rina Wati.', 0 bills
A7  GET /customers as a customer               -> 403, This page is for staff accounts.
A8  POST /logout, then GET /my-bills           -> 302, then 302 /login
```

A1 to A8 prove Problem 8.2 with its edge cases: the same e-mail twice, a wrong password, a customer on a staff page, a page after logout. Book: Section 8.4.4.

Worked Example: the Acceptance Run, Features

```
F0  GET /home as kasir1                          -> Outlet O01: 9 bills in the queue, 23 ready for
pickup.
F5  GET /services                                -> 200, 12 services
F1  GET /customers?q=Rahman                      -> 200, 6 customers
F2  POST /customers/new Bayu Saputra             -> Customer C0258 added.
F3  POST /customers/C0258/edit new phone         -> Customer C0258 updated.
F4  POST /customers/C0001/delete                 -> Customer C0001 has bills and cannot be deleted.
F4  POST /customers/C0258/delete                 -> Customer C0258 deleted.
F6  POST /bills/new C0257, S0001, 5 kg           -> Bill B2954 created: Rp 40.000.
F6  POST /bills/new with 0 kg                    -> The quantity must be a whole number from 1 to 100.
F8  POST /bills/B2954/pay Rp 20.000 cash         -> Payment of Rp 20.000 recorded. Balance Rp 20.000.
F8  POST /bills/B2954/pay Rp 30.000 qris         -> Payment refused: the amount must be between Rp 1 and
the balance.
F7  POST /bills/B2954/advance                    -> Bill B2954 is now finished.
F7  POST /bills/B2954/advance                    -> Bill B2954 is now departed.
F7  POST /bills/B2954/advance                    -> Bill B2954 cannot move forward.
E1  GET /nowhere                                 -> 404
F9  GET /my-bills as Rina                        -> B2954 | 2026-09-30 21:00:00 | Wash, dry & fold (per
kg) | departed | Rp 20.000
All 24 checks passed.
```

The refusals are checked too: the customer with bills, 0 kg, a payment larger than the balance, a bill that cannot move on. PHP, Java and Python printed the same lines; PHP also passed on MariaDB. `LAUNDRY_NOW` *freezes the clock, so the new bill is always* `B2954`*. Book: Section 8.4.4.*

The Write-Up

Table 8.4 The parts of the project write-up. Parts 4 and 5 need no code, and they are where a reader learns whether the team understood what it built.

Part	Contents	Where it comes from in this chapter
1. Database design	conceptual model (drawing and one sentence per relationship), physical model (drawing and SQL), the reasons for each key and type	Problem 8.1, Figure 8.1, Figure 8.2
2. Features	one row per feature: name, who, URLs, CRUD letters	Table 8.2
3. Evidence	for each feature: one screenshot, two sentences on what it shows, and the matching acceptance check	Problem 8.4
4. Analysis	what was measured, what worked, what did not and why, what you would change	the tempting wrong solutions of Problems 8.1 to 8.4
5. Conclusion	what the team knows now that it did not know before	your own
6. Contributions	each member, a percentage, and the concrete pieces of work behind it	the repository history

Write Table 8.4 while building. A contribution statement is believable only when it can be checked against the repository: "Putri 55%: the database, login, F6 to F8 and the acceptance test", not "Putri did the backend". Book: Section 8.4.4.

A Repository with No Secrets

```
--- without a .gitignore, git would add:
?? .env
?? data/laundry.db
?? env.example
?? php/lib.php
?? php/public/index.php
--- with the .gitignore of this problem:
?? .gitignore
?? env.example
?? php/lib.php
?? php/public/index.php
--- and git explains why .env is left out:
.gitignore:5:.env   .env
.gitignore:9:*.db   data/laundry.db
```

A secret pushed once stays in the history. `git log -p` shows it to anyone. The only repair is to *change* the secret.

Without a `.gitignore`, `git add .` would publish `.env` and 256 customers' personal data. Secrets come from the environment; `env.example` is committed with names and no values. Deploy through the host's environment settings, and finish with the acceptance test pointed at the public address. Book: Section 8.4.4.

Problem 8.5 — Takings per Outlet and Method

> **PROBLEM.** For September 2026, every outlet and payment method: the number of payments and the amount received, only combinations of at least Rp 400.000, by outlet and largest first.

Design, in the order the database works: `FROM Payment JOIN Bill`; `WHERE` filters *rows*; `GROUP BY` outlet and method; `HAVING SUM(p.amount) >= 400000` filters *groups*; `SELECT`, `ORDER BY`.

Half-open range: every moment from `2026-09-01` up to, but not including, `2026-10-01`.

`WHERE` may not use `SUM`, because when it runs there are no groups yet. Book: Section 8.5.1.

Problem 8.5 — The Query, and the Binding

Listing 8.8 The report query of `p85_report.sql`, with a half-open date range.

```
SELECT b.Outlet_id AS outlet, p.method, COUNT(*) AS payments, SUM(p.amount) AS received
  FROM Payment p
  JOIN Bill b ON b.id = p.Bill_id
 WHERE p.paid_at >= '2026-09-01' AND p.paid_at < '2026-10-01'   -- half-open: all of 30 September
 GROUP BY b.Outlet_id, p.method
HAVING SUM(p.amount) >= 400000
 ORDER BY b.Outlet_id, received DESC;
```

Listing 8.9 From `p85_report.php`: each placeholder bound with its proper type.

```
$st = db()->prepare($sql);
$st->bindValue(1, '2026-09-01');
$st->bindValue(2, '2026-10-01');
$st->bindValue(3, 400000, PDO::PARAM_INT);     // a number, not the text '400000'
$st->execute();
```

In Listing 8.9, `PDO::PARAM_INT` is not decoration: `execute([...])` sends text, and on SQLite every integer is "less than" any text, so Listing 8.8 came back empty with no error. MySQL converts and hides the problem. Book: Section 8.5.1.

Problem 8.5 — The Report

```
+--------+----------+----------+------------+
| outlet | method   | payments | received   |
+--------+----------+----------+------------+
| O01    | cash     |       51 | 1849000.00 |
| O01    | qris     |       39 | 1476000.00 |
| O01    | transfer |       10 |  472000.00 |
| O02    | cash     |       41 | 1809000.00 |
| O02    | qris     |       42 | 1714000.00 |
| O03    | qris     |       32 | 1810000.00 |
| O03    | cash     |       44 | 1700000.00 |
| O03    | transfer |       12 |  537000.00 |
+--------+----------+----------+------------+
```

Eight groups survive; the four `credit` groups and O02's transfers (Rp 302.000) are removed by `HAVING`. PHP, Java and Python printed the same report. Book: Section 8.5.1.

Problem 8.5 — `BETWEEN` Loses the Last Day

```
| 001    | cash     |       48 | 1750000.00 |
| 001    | qris     |       38 | 1431000.00 |
| 001    | transfer |        9 |  430000.00 |
| 002    | cash     |       41 | 1809000.00 |
| 002    | qris     |       42 | 1714000.00 |
| 003    | qris     |       31 | 1750000.00 |
| 003    | cash     |       41 | 1600000.00 |
| 003    | transfer |       12 |  537000.00 |
```

`BETWEEN '2026-09-01' AND '2026-09-30'` stops at the *first instant* of 30 September: 001 cash lost 3 payments and Rp 99.000, 003 QRIS one of Rp 60.000; in all nine payments worth Rp 346.000.

A report wrong by one day in thirty looks entirely plausible, which is what makes the mistake dangerous. Use `>= start AND < next day`, always. Book: Section 8.5.1.

Problem 8.6 — A Status Check That Cannot Be Tricked

Listing 8.10 From `p86_status.php`: the fixed SQL with two placeholders, and the tempting version that pastes the input into the text.

```
const SQL = "SELECT b.id, b.status, c.name FROM Bill b JOIN Customer c ON c.id = b.Customer_id
             WHERE b.id = ? AND SUBSTR(c.phone, -4) = ? ORDER BY b.id";

// The right way: the SQL text is fixed; the visitor's input travels separately, as data.
function check_safe(string $bill, string $phone4): ?array
{
    return row(SQL, [$bill, $phone4]);
}

// The tempting way: the visitor's input is pasted INTO the SQL text.
function check_unsafe(string $bill, string $phone4): ?array
{
    $sql = "SELECT b.id, b.status, c.name FROM Bill b JOIN Customer c ON c.id = b.Customer_id
            WHERE b.id = '$bill' AND SUBSTR(c.phone, -4) = '$phone4' ORDER BY b.id";
    return db()->query($sql)->fetch() ?: null;
}
```

A customer types a bill number and the last four digits of the phone; Listing 8.10 answers with the status and an initial and surname only. `SUBSTR(c.phone, -4)` *works in SQLite and MySQL. Book: Section 8.5.2.*

Problem 8.6 — Three Inputs, Two Versions

```
safe    B2953            '8079' -> B2953: queued (L. Mahendra)
safe    B2953            '0000' -> no bill with that number and phone
safe    x' OR 1=1 --     ''     -> no bill with that number and phone
unsafe  B2953            '8079' -> B2953: queued (L. Mahendra)
unsafe  B2953            '0000' -> no bill with that number and phone
unsafe  x' OR 1=1 --     ''     -> B0001: departed (I. Pratama)
```

The attack makes the unsafe text `… WHERE b.id = 'x' OR 1=1 -- ' AND SUBSTR(c.phone, -4) = '' …`: the quote closes the text, `OR 1=1` is true for every row, `--` comments out the phone check.

The unsafe page reveals `B0001`, a stranger's bill; in the safe version the same characters are an odd bill number that matches nothing. Escaping by hand must be remembered at every use; placeholders need no memory. Book: Section 8.5.2.

Problem 8.7 — A Cookie Is Not a Session

```
POST /wrong/login                              -> Set-Cookie: role=customer; path=/
GET /wrong/staff  Cookie: role=customer        -> 403
GET /wrong/staff  Cookie: role=staff           -> 200 Staff area: today's takings.
POST /right/login                              -> Set-Cookie: PHPSESSID=<random>; path=/; HttpOnly;
SameSite=Lax
GET /right/staff  with that session            -> 403
GET /right/staff  session + role=staff         -> 403
POST /wrong/login                              -> Set-Cookie: role=customer; Path=/
GET /wrong/staff  Cookie: role=customer        -> 403
GET /wrong/staff  Cookie: role=staff           -> 200 Staff area: today's takings.
POST /right/login                              -> Set-Cookie:
session=eyJyb2xlIjoiY3VzdG9tZXIifQ.<time>.<signature>; HttpOnly; Path=/
  the payload decodes to                       -> {"role":"customer"}
GET /right/staff  with that session            -> 403
GET /right/staff  forged payload               -> 403
```

A cookie is a claim made by the visitor, never a fact: `Cookie: role=staff` opens the wrong staff page. A session keeps the fact on the server. Flask's signed session can be read (the payload is Base64 `{"role":"customer"}`) but not changed: the forged payload gets 403. Book: Section 8.5.3.

Problem 8.8 — A Day's Bills as XML

Listing 8.11 The tree-building loop of `p88_export.php`: DOM writes the tags and escapes every character.

```
foreach ($bills as $b) {
    $bill = $root->appendChild($doc->createElement('bill'));
    $bill->setAttribute('id', $b['id']);
    $bill->setAttribute('status', $b['status']);
    $cust = $bill->appendChild($doc->createElement('customer'));
    $cust->setAttribute('id', $b['cid']);
    $cust->appendChild($doc->createTextNode($b['name']));
    $svc = $bill->appendChild($doc->createElement('service'));
    $svc->setAttribute('id', $b['sid']);
    $svc->appendChild($doc->createTextNode($b['description']));     // "&" becomes "&amp;" here
    $bill->appendChild($doc->createElement('unit', (string) $b['unit']));
    $charged = $bill->appendChild($doc->createElement('charged', (string) (int) $b['payment']));
    $charged->setAttribute('currency', 'IDR');
}
```

In Listing 8.11 identifiers are attributes and readable things are content; the amount is a plain integer with `currency="IDR"`. Nowhere does the program write `&`: `saveXML()` does. Book: Section 8.6.1.

Problem 8.8 — The Export, and the Glued Version

```
<?xml version="1.0" encoding="UTF-8"?>
<bills outlet="O03" date="2026-09-30">
  <bill id="B2944" status="queued">
    <customer id="C0015">Nurul Suryadi</customer>
    <service id="S0001">Wash, dry &amp; fold (per kg)</service>
    <unit>5</unit>
    <charged currency="IDR">40000</charged>
  </bill>
  <bill id="B2950" status="queued">
    <customer id="C0088">Ma'ruf Saputra</customer>
    <service id="S0001">Wash, dry &amp; fold (per kg)</service>
    <unit>6</unit>
    <charged currency="IDR">48000</charged>
Glued string well-formed? no: xmlParseEntityRef: no name (column 76)
```

Wash, dry & fold becomes `&` and Ma'ruf needs nothing. The glued string meets `& fold` at column 76 and the parser refuses the whole document: head office receives nothing. It passes every test on a day with no `&` in a description. Book: Section 8.6.1.

Problem 8.9 — Valid, Not Merely Well-Formed

Listing 8.12 The content model of a `bill` in `bills.xsd`: named types, a positive integer, and a currency fixed to `IDR`.

```
<xs:sequence>
  <xs:element name="customer" type="Named"/>
  <xs:element name="service" type="Named"/>
  <xs:element name="unit" type="xs:positiveInteger"/>
  <xs:element name="charged">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:nonNegativeInteger">
          <xs:attribute name="currency" type="xs:string" fixed="IDR" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:sequence>
```

The rest of Listing 8.12: `Status` is an enumeration of three words, the bill id matches `B[0-9]{4}`, and `date` is an `xs:date`. Book: Section 8.6.2.

Problem 8.9 — Three Mistakes Found

Listing 8.13 `bills-broken.xml`: well-formed, and wrong in three places.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bills outlet="O03" date="2026-09-30">
  <bill id="B2944" status="lost">
    <customer id="C0015">Nurul Suryadi</customer>
    <service id="S0001">Wash, dry &amp; fold (per kg)</service>
    <unit>five</unit>
    <charged currency="USD">40000</charged>
  </bill>
</bills>
```

```
bills-O03-2026-09-30.xml   well-formed: yes  valid: yes
bills-broken.xml           well-formed: yes  valid: no
  line 3: Element 'bill', attribute 'status': [facet 'enumeration'] The value 'lost' is not an element
of the set {'queued', 'finished', 'departed'}.
  line 6: Element 'unit': 'five' is not a valid value of the atomic type 'xs:positiveInteger'.
  line 7: Element 'charged', attribute 'currency': The value 'USD' does not match the fixed value
constraint 'IDR'.
```

Listing 8.13 parses: well-formed. The schema finds `lost`, `five` and `USD`. Stopping after the parser would book forty thousand US dollars as rupiah. Python's lxml prints the same; Java's `ErrorHandler` collects every error. Book: Section 8.6.2.

Problem 8.10 — Prices That Are Numbers

Listing 8.14 From `p810_prices.php`: the price cast to an integer, and `json_encode` writing the array.

```
$cast = !in_array('--no-cast', $argv, true);
$services = [];
foreach (rows('SELECT id, description, price FROM Service ORDER BY id LIMIT 3') as $r) {
    $services[] = ['id' => $r['id'], 'description' => $r['description'],
                   'price' => $cast ? (int) $r['price'] : $r['price']];   // MySQL hands DECIMAL over as
text
}
echo json_encode(['currency' => 'IDR', 'services' => $services],
                 JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES | JSON_UNESCAPED_UNICODE), "\n";
```

Listing 8.15 `p810_client.js`: a client that does arithmetic with the prices it receives.

```
// p810_client.js - Problem 8.10: a JavaScript client adds two prices from the JSON it receives.
// Run: php ../php/p810_prices.php | node p810_client.js
const data = JSON.parse(require("fs").readFileSync(0, "utf8"));   // 0 = standard input
const [wash, , iron] = data.services;
console.log("typeof price:", typeof wash.price);
console.log("wash + iron, per kg:", wash.price + iron.price);
```

Listing 8.14 with the cast; Listing 8.15 adds the price of washing to the price of ironing. Python's `int(p)` matters most with PyMySQL, whose `Decimal` `json.dumps` refuses. Book: Section 8.6.3.

Problem 8.10 — Without the Cast, on MySQL

```
{
    "currency": "IDR",
    "services": [
        {
            "id": "S0001",
            "description": "Wash, dry & fold (per kg)",
            "price": "8000.00"
typeof price: string
wash + iron, per kg: 8000.006000.00
```

PDO returns `DECIMAL` from MySQL as text, `json_encode` writes a string, and JavaScript's `+` joins strings: `8000.006000.00`. With the cast: `typeof price: number` and `14000`.

Multiplication hides the problem (`"8000.00" * 2` is 16000), which is why such bugs survive until the first addition. Book: Section 8.6.3.

Problem 8.11 — The Queue Page as MVC

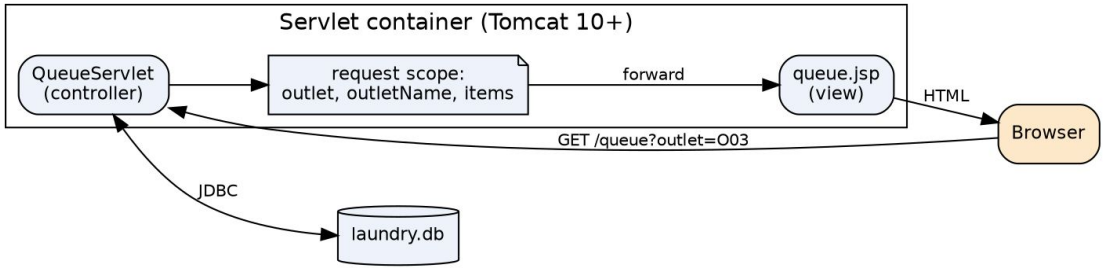


Figure 8.5 The queue page in the MVC arrangement: the servlet reads and decides, the model travels in request scope, the JSP page only displays.

Figure 8.5: `QueueServlet` checks the outlet, reads the bills with JDBC, puts `outlet`, `outletName` and `items` in request scope, and forwards to `/WEB-INF/queue.jsp`, which no browser can open directly. Book: Section 8.7.1.

Problem 8.11 — The Controller

Listing 8.16 The end of `QueueServlet.doGet`: rows become `QueueItem` beans, the model goes into request scope, and the request is forwarded.

```
            try (PreparedStatement st = c.prepareStatement(SQL)) {
                st.setString(1, outlet);
                ResultSet rs = st.executeQuery();
                while (rs.next()) {
                    items.add(new QueueItem(rs.getString(1), rs.getString(2), rs.getString(3),
                            rs.getString(4), rs.getString(5), rs.getInt(6)));
                }
            }
        } catch (SQLException e) {
            throw new ServletException(e);
        }
        req.setAttribute("outlet", outlet);                 // the model, in request scope
        req.setAttribute("outletName", outletName);
        req.setAttribute("items", items);
        req.getRequestDispatcher("/WEB-INF/queue.jsp").forward(req, resp);
```

In Listing 8.16 the query orders by `b.status <> 'finished'`, so finished bills come first; an unknown outlet gets `sendError(404, …)`. The servlet writes no HTML. Book: Section 8.7.1.

Problem 8.11 — The View

Listing 8.17 The loop of `queue.jsp`: Expression Language to read, JSTL to loop, `c:out` to escape.

```
<c:forEach var="b" items="${items}">
<tr>
  <td><c:out value="${b.id}"/></td>
  <td><c:out value="${b.status}"/></td>
  <td><c:out value="${b.rack}"/></td>
  <td><c:out value="${b.customer}"/></td>   <%-- c:out escapes < > & ' " --%>
  <td><c:out value="${b.service}"/></td>
  <td>${b.unit}</td>
</tr>
</c:forEach>
```

> **`<td>${b.customer}</td>` does not escape.** The day a customer registers as `<script>alert('hi')</script>`, every queue screen runs it. `|safe` in Jinja is the same hole.

`${b.customer}` in Listing 8.17 calls `getCustomer()`, which is why the bean needs getters. `${b.unit}` is bare because it is an `int`. Book: Section 8.7.1.

Problem 8.11 — Run in a Stand-In Container

```
browser A: GET {outlet=O03}
  forward to /WEB-INF/queue.jsp with request attributes:
    outlet = O03
    outletName = Profitina Laundry Wiyung
    items = 10 items:
      B1097 finished C04 Rafi Susanto, Ironing only (per kg) x 3
      B1321 finished B14 Nadia Zulkarnain, Wash, dry & fold (per kg) x 6
      B1542 finished C05 Farah Putri Kusuma, Dry cleaning - dress/kebaya (per item) x 2
      B2403 finished C11 Winda Firmansyah, Wash, dry & fold (per kg) x 10
      B2938 finished A05 Syaiful Wijaya, Wash, dry & fold (per kg) x 5
      B2940 queued C06 Susanti Mahendra, Bed cover, any size (per item) x 1
      B2941 queued C18 Farah Wijaya, Wash, dry & fold (per kg) x 8
      B2944 queued A01 Nurul Suryadi, Wash, dry & fold (per kg) x 5
      B2950 queued A02 Ma'ruf Saputra, Wash, dry & fold (per kg) x 6
      B2952 queued B07 Khadijah Zulkarnain, Wash, dry & fold (per kg) x 5
```

No container could be installed on the build machine, so the servlet was compiled against the official API jar and run in `StubContainer.java`, which prints the model the JSP page would receive. The Flask and JDK versions produced the same page, byte for byte. Book: Section 8.7.1.

Problem 8.12 — Four Places to Keep a Number

Table 8.5 Four places to keep a value in a servlet. Only the session is private to one visitor.

Where	Created	Destroyed	Shared by
a local variable inside `doGet`	at every request	at the end of that request	nobody
a field of the servlet	once, when the container creates the servlet	when the servlet is destroyed	every visitor and every thread
a session attribute	at the visitor's first `getSession()`	at logout or time-out	one visitor's requests
an application (context) attribute	when set, usually in `init()`	when the application stops	every visitor

Table 8.5: the container creates one servlet object and sends every request from every visitor to it, often on several threads at once. Book: Section 8.7.2.

Problem 8.12 — The Servlet

Listing 8.18 From `VisitServlet.doGet`: a local variable, a field, a session attribute and an application attribute.

```
        int local = 0;
        local++;                                        // a local variable is born and dies per
request
        synchronized (this) {
            fieldCount++;                               // shared by all visitors (and all threads)
        }
        HttpSession session = req.getSession();         // this visitor only
        Integer mine = (Integer) session.getAttribute("mine");
        mine = (mine == null) ? 1 : mine + 1;
        session.setAttribute("mine", mine);
        ServletContext app = getServletContext();       // the whole application
        int total;
        synchronized (app) {
            total = (Integer) app.getAttribute("total") + 1;
            app.setAttribute("total", total);
        }
```

In Listing 8.18 both shared counters are updated inside `synchronized` blocks, or two threads could read the same old value and lose a visit. `init()` sets the application attribute to 0. Book: Section 8.7.2.

Worked Example: Rina, Rina, Bayu, Rina

```
init(): VisitServlet created
Rina: GET {}
  local=1 field=1 session=1 application=1
Rina: GET {}
  local=1 field=2 session=2 application=2
Bayu: GET {}
  local=1 field=3 session=1 application=3
Rina: GET {}
  local=1 field=4 session=3 application=4
destroy(): VisitServlet removed after 4 requests
```

local is 1 every time; **field** and **application** count everybody: 1, 2, 3, 4; only **session** answers "your visits": Rina 1, 2, 3 and Bayu 1.

The tempting wrong solution keeps "your visits" in a field, and Bayu's first visit shows 3. In a JSP page the same mistake is `<%! int count = 0; %>`*. Book: Section 8.7.2.*

The Problems Compared, 8.1 to 8.6

Table 8.6 Problems 8.1 to 8.6: what each practices and the case its tempting solution gets wrong.

Problem	Topic	Main idea	What the tempting solution gets wrong
8.1	database design	entities, keys, enforced foreign keys	SQLite without `PRAGMA foreign_keys = ON` stores orphans
8.2	authentication	salted slow hash, new session id at login	an unsalted fast hash reveals equal passwords
8.3	CRUD features	front controller, transactions, server-side checks	trusting the browser's `min="1"`
8.4	evidence and delivery	acceptance test over HTTP, `.gitignore`, environment settings	a secret committed once stays in the history
8.5	SQL reports	`WHERE` for rows, `HAVING` for groups, half-open ranges	`BETWEEN` loses the last day; text-bound numbers
8.6	SQL injection	placeholders keep input out of the SQL text	`' OR 1=1 --` reveals a stranger's bill

Read down the last column of Table 8.6: each tempting solution works on the ordinary example and fails on one particular input. Book: Section 8.8.

The Problems Compared, 8.7 to 8.12

Table 8.7 Problems 8.7 to 8.12: what each practices and the case its tempting solution gets wrong.

Problem	Topic	Main idea	What the tempting solution gets wrong
8.7	cookies and sessions	facts stay on the server; signed is not secret	a `role` cookie is edited to `staff`
8.8	XML output	build a tree, let the library escape	`&` in a description breaks the document
8.9	XML Schema	well-formed, then valid	a well-formed `lost`, `five`, `USD` is accepted
8.10	JSON	numbers as numbers	MySQL `DECIMAL` becomes a string; `+` joins text
8.11	servlet and JSP	controller fills request scope, view only displays	bare `${…}` does not escape
8.12	lifecycle and scopes	one servlet object, many visitors	a field counts everybody's visits

Table 8.7 is the strongest argument for Step 6: write the edge cases down before writing the code. Book: Section 8.8.

Profitina in Practice

Accounts and personal data: Argon2id hashes, rate-limited logins, export and deletion under UU PDP 27/2022; each feature a group of routes with a permission check in front.

One database, two engines: PostgreSQL in production, SQLite in parts of the test path, with a guard; Problem 8.5 showed why.

Structured data for machines: schema.org, `robots.txt`, `/llms.txt`: well-formed, schema-true, rightly typed.

Testing as a release gate: well over a thousand tests; the acceptance test of Problem 8.4 at chapter size.

The product advises and its users decide; the runnable version of every idea is the Profitina Laundry code. Book: Section 8.9.

Common Mistakes

Design and data: pages before tables; a foreign key that is not enforced; money as a float or a string.

Accounts: reversible or fast unsalted password storage; the same session id across a login; trusting a cookie.

Requests: checking only in the browser; changing data with `GET`; two related writes without a transaction; a secret in the repository.

SQL, XML, JSON, JSP: `BETWEEN` on timestamps; a number bound as text; SQL from input; strings glued into XML or JSON; stopping at well-formed; printing without escaping; per-visitor data in a servlet field.

Each mistake is listed in the book with its fix and the problem that shows it. Book: Section 8.10.

Summary: Five Things to Remember

1. Solve a web problem in eight steps: data first, then URLs, permissions and tests, and only then code.

2. Design at two levels, conceptual and physical; a foreign key is only as good as its enforcement.

3. Store slow salted hashes, keep identity on the server, regenerate the session id at login, and redirect after every successful `POST`.

4. Filter rows with `WHERE` and groups with `HAVING`, use half-open ranges, bind every value with its type; write XML and JSON with libraries and validate what arrives.

5. A servlet fills request scope and forwards to a view that only displays and always escapes; per-visitor data belongs in the session.

Three applications in three languages on two database systems agree to the rupiah. Book: Section 8.11.

Next Time

Chapter 9 — ASP.NET and Rich Internet Applications. Ajax, the .NET platform, Razor Pages on .NET 10, and Web Forms shown for concept.

The live bill status of Profitina Laundry as an Ajax feature, with the same contract served by ASP.NET Core, PHP, Java and Python.

Before then: work Exercises 8.1, 8.2, 8.5, 8.7 and 8.10, and install the .NET 10 SDK.

Solutions to every exercise, with the real output of each query, are in the book. Book: Sections 8.12 and 8.13.

Lecture 9 — ASP.NET and Rich Internet Applications

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 9

Lecturer: Irfan Subakti — yifana@gmail.com

Today a web page learns to change one piece of itself without reloading, and we meet the last server technology of the course: ASP.NET Core on .NET 10. Book: Chapter 9, Sections 9.1–9.17.

What This Meeting Covers

1. Rich internet applications — what “rich” means, the plugin years, and why HTML5 won.

2. Ajax — asking for *data* instead of a page: `XMLHttpRequest`, `fetch()`, promises and JSON.

3. .NET and ASP.NET — from Web Forms in 2002 to ASP.NET Core on .NET 10 LTS; installing the SDK.

4. Razor Pages — `Program.cs`, pages and page models, Razor syntax, handlers, binding, validation, tag helpers.

5. Web Forms for concept — server controls, postback, view state, the UpdatePanel, and their modern forms.

6. Profitina Laundry — a live bill status with ETags and polling, one JavaScript front end, four servers.

The two halves meet in the Laundry feature: an Ajax page served by Razor Pages, then by PHP, Java and Python. Book: Section 9.1.

Where We Left Off: Click, Wait, New Page

Chapter 1: **a browser asks, a server answers**. Chapters 2 to 7 filled in both sides: HTML, CSS, PHP, MySQL, XML, servlets and JSP.

In every chapter so far the rhythm was the same: the visitor clicks, the server builds a **complete new page**, the browser throws the old one away.

A desktop program does not behave like that: press a button and **one number changes** while everything else stays where it was.

Today's two questions: **how can a web page feel like a desktop program?** and **what is ASP.NET**, the server framework behind many banks, offices and factories?

Sections 9.2 and 9.3 answer the first, Sections 9.4 to 9.9 the second, and Section 9.10 joins them. Book: Section 9.1.

What “Rich” Means

> **DEFINITION.** A **rich internet application** (RIA), also called a *rich web application*, is a web application with many of the characteristics of desktop software: immediate response, parts of the screen that update on their own, drag and drop, editing in place. It is delivered through the browser and needs no installation.

Desktop application: instant, but installed and updated on every machine. **Web application:** nothing to install, fixed for everybody at once — but classically a round trip and a full page per action.

A test of richness: when one thing changes, what happens to the rest of the page? In a rich one it stays exactly where it was, including what you typed and where you scrolled.

An RIA tries to have both sets of virtues. The single-page application is a relative, separated from it in Section 9.3.7. Book: Section 9.2.1.

Three Circles Overlap

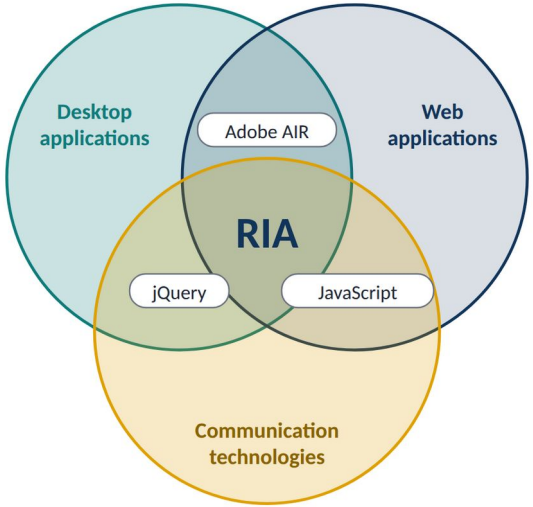


Figure 9.1 A rich internet application sits where desktop applications, web applications and communication technologies overlap. The labels in the middle name three ways of building one: Adobe AIR, jQuery and plain JavaScript.

The third circle of Figure 9.1, communication technologies, is the one people forget: an RIA is rich because it talks to the server in the background, in small pieces, while the user keeps working. Book: Section 9.2.1.

Before HTML5: the Age of Plugins

Table 9.1 The main RIA technologies before HTML5 and their fate. Every one that needed a plugin inside the browser is now gone from it.

Technology	From	What it was	What became of it
Adobe Flash, Apache Flex	Macromedia, then Adobe	A plugin running ActionScript; Flex added MXML and a class library for business screens	End of life 31 December 2020; browsers refuse to run it
Adobe AIR	Adobe	The Flash runtime outside the browser (TweetDeck, eBay Desktop, Pandora One, *Machinarium*)	Handed to Harman in 2019; a few games remain
Java applets	Sun, then Oracle	A Java program in a rectangle of the page; faster than the JavaScript of its day	Deprecated in Java 9 (2017); the plugin was removed in JDK 11 (2018)
JavaFX	Sun, then Oracle	A Java toolkit for rich interfaces, first with its own script language	Lives on as OpenJFX for desktops; no longer in browsers
Microsoft Silverlight	Microsoft	A .NET plugin programmed in XAML and C#	Support ended 12 October 2021
XForms	W3C	A declarative, XML-based successor to HTML forms	A standard used in some XML tools; never built into browsers
OpenLaszlo	Laszlo Systems	An XML-and-JavaScript language, LZX, compiled to Flash or to browser code	Discontinued

Table 9.1: a plugin was a separate program installed into the browser that took over a rectangle of the page. Flash was the most successful; applets were older; Silverlight was Microsoft's answer to Flash. Book: Section 9.2.2.

Where the Plugins Went

Mobile. The iPhone (2007) never ran Flash; when many visitors could not see a Flash page at all, building one stopped making sense.

Security. A plugin runs whatever a page gives it; Flash, Java and Silverlight were among the most attacked desktop programs until browsers blocked and removed them.

HTML5 made them unnecessary: canvas and WebGL, audio and video, storage, workers, WebSockets, `XMLHttpRequest` — built into every browser, owned by no company.

YouTube: a Flash application in 2005, HTML5 by default in January 2015, streaming with MPEG-DASH over plain HTTP.

> **THE KEY INSIGHT.** Every RIA technology that required a plugin inside the browser has died, whatever its quality and whoever stood behind it. What survived is what the browser itself understands: HTML, CSS, JavaScript and HTTP.

Since 2019 there is one HTML Living Standard, maintained by the WHATWG and updated continuously. Book: Sections 9.2.3 and 9.2.4.

Two Histories Side by Side

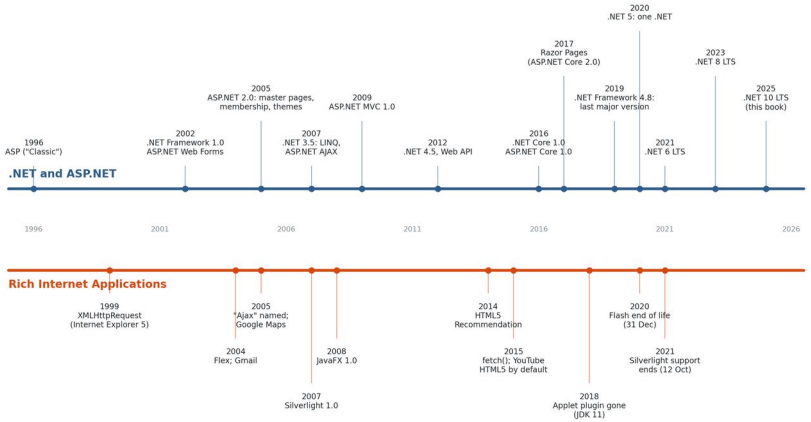


Figure 9.2 Two timelines. Above: .NET and ASP.NET, from classic ASP in 1996 to .NET 10 LTS in 2025. Below: rich internet applications, from `XMLHttpRequest` in 1999 through the plugin years to the end of Flash (2020) and Silverlight (2021).

Figure 9.2 is the map of this whole meeting: the .NET line above, from classic ASP to .NET 10 LTS, and the RIA line below, from `XMLHttpRequest` in 1999 to the end of Flash and Silverlight. Book: Section 9.2.3.

Ajax: RIA Without a Plugin

> **DEFINITION. Ajax** is the technique of sending an HTTP request from JavaScript while a page is displayed, without leaving the page, and using the answer to change part of the page through the DOM. *Asynchronous* means that the script does not stop and wait: the page stays usable, and a function runs later when the answer arrives.

Named in February 2005 by Jesse James Garrett, after Gmail (2004) and Google Maps (2005): *Asynchronous JavaScript and XML*.

In the classic model the user's activity **stops dead** at every request; in the Ajax model it never stops — that gap is the whole difference.

Some inputs are answered by the engine at once, in the browser; others go to the server, which works while the user continues. Book: Sections 9.3.1 and 9.3.2.

The Classic Model and the Ajax Model

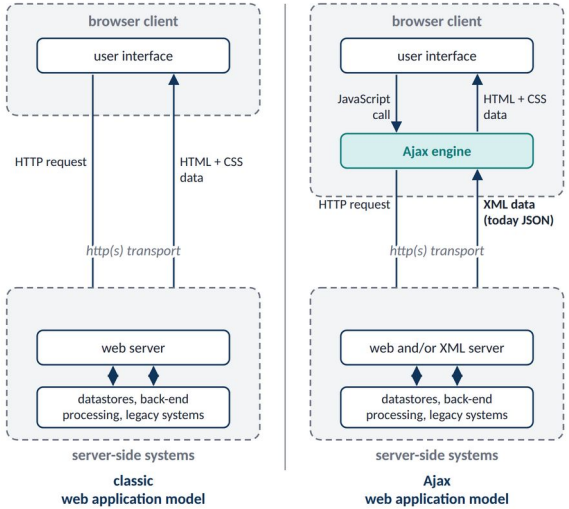


Figure 9.3 The classic web application model (left) and the Ajax model (right). In the Ajax model an *Ajax engine* sits between the user interface and the server; the interface calls it with JavaScript, and it exchanges data — originally XML, today JSON — with the server. Redrawn after J. J. Garrett (2005).

On the left of Figure 9.3, every chapter so far: HTTP request, whole page back. On the right, one box added inside the browser, the Ajax engine; what travels is data — XML in 2005, JSON today. Book: Section 9.3.1.

Asking in the Style of 2005

Listing 9.1 `xhr-status.js`: asking for a bill's status with `XMLHttpRequest`, in the style of 2005. `send()` returns immediately; the function in `onreadystatechange` runs later, when the answer is complete (`readyState` 4).

```
// xhr-status.js - the question the status page asks, in the style of 2005:
// XMLHttpRequest and a callback. Load it into a page served by any track, then call
// checkStatus("B2952", "0812-0000-4710", function (code, data) { ... }).
function checkStatus(bill, phone, done) {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "/api/status?bill=" + encodeURIComponent(bill) +
                  "&phone=" + encodeURIComponent(phone), true);   // true = asynchronous
  xhr.setRequestHeader("Accept", "application/json");
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {                                    // 4 = DONE
      done(xhr.status, xhr.status === 200 ? JSON.parse(xhr.responseText) : null);
    }
  };
  xhr.send();                                                      // returns at once
}
```

Listing 9.1: `open` *prepares the request (*`true` *= asynchronous),* `send()` *returns at once, and the* `onreadystatechange` *function runs when the answer is complete. Called in Chromium against the Flask track it gave* `200 B2952 In process 5 kg` *for the right phone and* `404 null` *for a wrong one. Book: Section 9.3.2.*

Thin Client and Fat Client

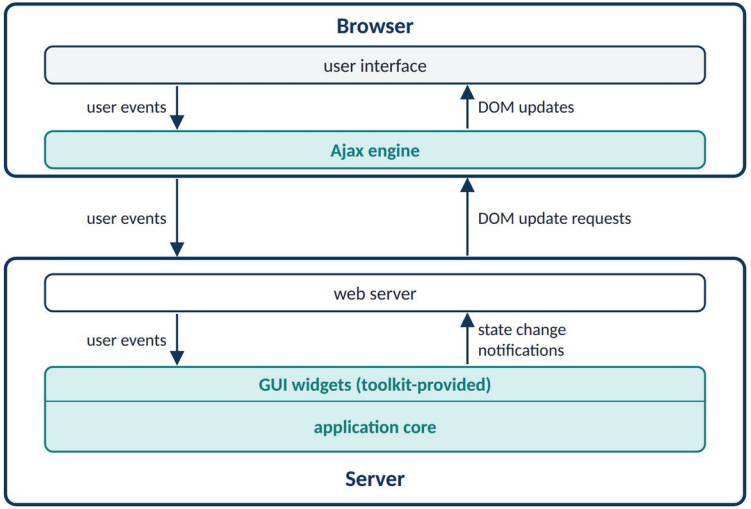


Figure 9.4 The Ajax *thin client*. The browser holds only the user interface and a small Ajax engine that forwards user events and applies the DOM updates it is told to make; the GUI widgets and the application core live on the server.

In the thin client of Figure 9.4 the server keeps the state of every visitor's screen. In the fat client the widgets run in the browser and call a web service only for data. Thin and fat are two ends of a slider; the Laundry page sits near the thin end. Book: Section 9.3.3.

fetch() and Promises

Listing 9.2 `fetch-status.mjs`: the status question with `fetch()`, runnable in Node.js 18 or newer. The second request sends the version it already has in `If-None-Match`.

```
// fetch-status.mjs - the same question with fetch(), from Node.js 18 or newer:
//     node fetch-status.mjs http://127.0.0.1:8940
const base = process.argv[2] ?? "http://127.0.0.1:8940";
const url = new URL("/api/status", base);
url.searchParams.set("bill", "B2952");
url.searchParams.set("phone", "0812-0000-4710");

const first = await fetch(url, { headers: { Accept: "application/json" } });
const etag = first.headers.get("ETag");
const data = await first.json();
console.log(first.status, etag, data.bill_id, data.status_text, data.balance);

const again = await fetch(url, { headers: { "If-None-Match": etag } });
console.log(again.status, again.statusText, "- body length", (await again.text()).length);
```

`fetch()` returns a promise; `await` pauses that function only, never the page. Listing 9.2 asks twice: the first answer was `200 "B2952-1-1"` with the data, the second `304` with an empty body, because the request carried the version it already had. Book: Section 9.3.5.

JSON: the Contract

Table 9.2 The JSON contract of the live bill status, extending `GET /api/status` of the Laundry API (`App/docs/api.md`) with `quantity` and `history`. There is no customer name, address or e-mail.

Field	Type	Meaning
`bill_id`	string	Bill number, upper case: `"B2952"`
`status`	string	`queued`, `finished` or `departed`, as stored in `Bill.status`
`status_text`	string	What a customer reads: *In process*, *Ready for pickup*, *Collected*, *Cancelled*
`arrived`, `finished`, `departed`	string or `null`	Shop time `YYYY-MM-DD HH:MM:SS` (WIB); `null` if it has not happened
`promised`	string	When the laundry should be ready, Equation (9.3)
`service`, `unit`, `quantity`	string, number, string	The service, the units charged, and both in words: `"5 kg"`
`payment`, `balance`	number	Whole rupiah charged, and still to pay, Equation (9.4)
`outlet`	object	`name` and `phone` of the outlet holding the laundry
`history`	array	One object per status event — `status`, `text`, `at` — oldest first

Table 9.2 is the contract between the four servers of this meeting and the one front end. No name, address or e-mail: UU PDP No. 27/2022 requires data minimization for a page anyone with a bill number and a phone can open. Book: Section 9.3.6.

Three Ways to Build a Web Application

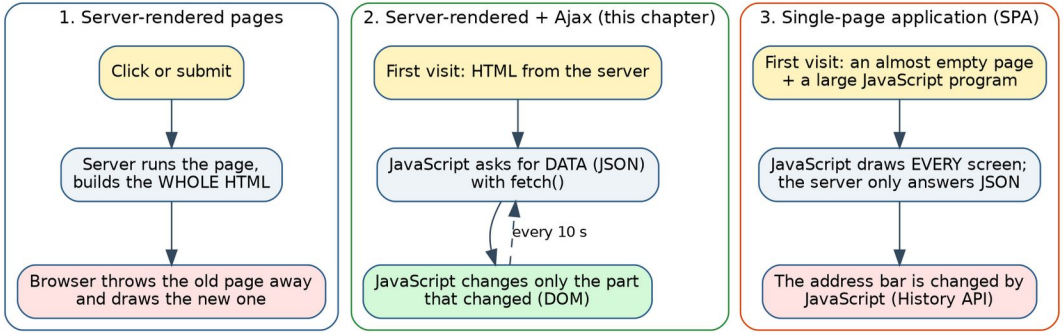


Figure 9.5 Three ways to build a web application. (1) The server builds every page. (2) The server builds the pages and JavaScript refreshes parts of them with data — this chapter. (3) A single-page application: JavaScript builds every screen and the server only answers with JSON.

Figure 9.5: (1) server-rendered pages, everything so far; (2) server-rendered pages with Ajax, this meeting; (3) a single-page application, where JavaScript draws every screen. Book: Section 9.3.7.

Choosing Among the Three

Table 9.3 The three ways compared. The middle column is the sensible default for most business applications, including Profitina Laundry.

	Server-rendered	Server-rendered + Ajax	Single-page application
First screen	Fast: HTML arrives ready	Fast: HTML arrives ready	Slower: the program must load and run first
After a click	A whole new page	Often only a piece of data	Only data
Works without JavaScript	Yes	Yes, with less comfort	No
State of the screen lives	On the server	Mostly on the server	In the browser
The team must master	One server language	One server language + some JavaScript	A JavaScript framework + an API server
Good for	Documents, forms, shops, back offices	The same, with live parts	Editors, maps, dashboards used for hours

Table 9.3: the middle column keeps the page working without JavaScript — progressive enhancement. An SPA moves work, state and bugs into the browser; choose it for screens people use for hours. Book: Section 9.3.7.

What .NET Is

> **DEFINITION.** The **Common Language Runtime** (CLR) is the virtual machine of .NET: it loads programs compiled to *intermediate language* (IL), compiles the IL to machine code just in time, collects garbage and enforces type safety. The **class library** is the large set of ready-made classes every .NET program can use. A .NET *language* — C#, F#, VB.NET — is any language whose compiler produces IL.

The comparison with Chapter 7 is exact: CLR for the JVM, IL for bytecode, the class library for Java's, **C# for Java**.

Before it: CGI (a program per request) and classic ASP (VBScript in pages, interpreted, Windows only). ASP.NET (2002) **compiled** pages into classes of a strongly typed language.

The original .NET Framework ran only on Windows, and that fact shaped the next decade. Book: Sections 9.4.1 and 9.4.2.

From ASP.NET to ASP.NET Core and One .NET

2002–2010: Web Forms on the .NET Framework; 2.0 added master pages and membership, 3.5 ASP.NET AJAX and MVC, 4.0 routing and the encoding syntax `<%: %>`.

2016: ASP.NET Core 1.0 and .NET Core — lean, modular (NuGet), open source, cross-platform, with its own web server, **Kestrel**. **2017:** Razor Pages.

2019: the .NET Framework frozen at 4.8. **2020:** .NET 5 drops "Core" and becomes *the* .NET; a new version every November.

> **DEFINITION.** Each .NET release is either **LTS** (*long-term support*, three years; even-numbered versions) or **STS** (*standard-term support*, two years since .NET 9; odd-numbered versions). **.NET 10**, released in November 2025, is the current LTS, supported until November 2028.

Razor Pages (2017) sits beside Web Forms, MVC, minimal APIs and Blazor in the family; it is closest to what we know: a file answering its own address, like PHP, with `OnGet` and `OnPost` like a servlet's `doGet` and `doPost`. Book: Sections 9.4.3–9.4.5.

Installing and Checking the SDK

Install the **SDK**, not only the runtime: Windows `winget install Microsoft.DotNet.SDK.10`; macOS the `.pkg` or Homebrew; Ubuntu `sudo apt install dotnet-sdk-10.0`. Then, in VS Code, the **C# Dev Kit**.

```
dotnet --version
dotnet --list-sdks
dotnet --list-runtimes
```

Listing 9.3 The three checks and their answers on the build machine: SDK 10.0.112 with runtimes 10.0.12. Your patch numbers and folders will differ; the first two digits must be `10.0`.

```
10.0.112
10.0.112 [/usr/lib/dotnet/sdk]
Microsoft.AspNetCore.App 10.0.12 [/usr/lib/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.NETCore.App 10.0.12 [/usr/lib/dotnet/shared/Microsoft.NETCore.App]
```

Listing 9.3: the first two digits must be `10.0`. `Microsoft.AspNetCore.App` is the shared framework — Kestrel, routing, Razor Pages, JSON, antiforgery — so a web project needs no download. Book: Sections 9.5.1–9.5.3.

One Command, One Web Application

```
dotnet new webapp -o FirstApp
cd FirstApp
dotnet run --urls http://127.0.0.1:8901
```

Listing 9.4 The log of `dotnet run --urls http://127.0.0.1:8901` (three earlier lines, about the launch settings and data-protection keys, omitted). The warning is harmless: the template redirects HTTP to HTTPS and was given no HTTPS address.

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:8901
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /…/FirstApp
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
```

`dotnet new webapp -o FirstApp` creates the project; `dotnet run` restores, compiles and starts it. Listing 9.4: the application is the web server (Kestrel), it starts in about a second, and the environment is Development. Book: Section 9.6.1.

The Anatomy of a Razor Pages Project

```
FirstApp/
├── FirstApp.csproj            the project: what to build, for which .NET
├── Program.cs                 the program: services, pipeline, run
├── appsettings.json           settings (appsettings.Development.json overrides them)
├── Properties/
│   └── launchSettings.json    how "dotnet run" and F5 start it
├── Pages/                     one .cshtml (+ .cshtml.cs) per page
│   ├── Index.cshtml           the page at /
│   ├── Index.cshtml.cs        its page model
│   ├── Privacy.cshtml(.cs), Error.cshtml(.cs)
│   ├── _ViewImports.cshtml    usings and tag helpers for every page
│   ├── _ViewStart.cshtml      runs before every page: chooses the layout
│   └── Shared/_Layout.cshtml  the frame around every page
└── wwwroot/                   static files: css/, js/, lib/, favicon.ico
```

The `Pages` folder is the site map: a file's path under `Pages` is its address. `wwwroot` plays the part of Apache's document root; `Program.cs` the part of `web.xml` and the container of Chapter 7. Book: Section 9.6.2.

Program.cs: the Services

Listing 9.5 The first lines of the template's `Program.cs`: create the builder, register the services, build the application.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();
```

No class and no `Main`: C# allows *top-level statements* in one file.

`CreateBuilder(args)` reads the configuration (the `appsettings` files, environment variables, the command line), sets up logging and prepares a **service container**.

A *service* is any object the application may ask for later; the container creates it and hands it over — **dependency injection**.

Listing 9.5: `AddRazorPages()` registers what Razor Pages needs. Every ASP.NET Core program has the same three parts: services, pipeline, run. Book: Section 9.6.3.

Program.cs: the Pipeline

Listing 9.6 The rest of the template's `Program.cs`: each `Use…` or `Map…` line adds middleware, in pipeline order; `app.Run()` starts Kestrel.

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();
app.MapRazorPages()
   .WithStaticAssets();

app.Run();
```

Listing 9.6 is the filter chain of Chapter 7 in another form. In Development a detailed error page with the stack trace is shown — a gift to an attacker; a published application runs as Production. Book: Section 9.6.3.

A Page and Its Page Model

Listing 9.7 `Pages/Index.cshtml` of the template. `@page` makes the file a page with its own address; `@model IndexModel` names its page model; `@{ … }` runs C# before the HTML is written.

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://learn.microsoft.com/aspnet/core">building Web apps with ASP.NET
Core</a>.</p>
</div>
```

`@page` gives the file its address: `Pages/Index.cshtml` answers `/`, `Pages/Privacy.cshtml` answers `/Privacy`.

Per request ASP.NET Core creates a **fresh** page model, calls the matching handler (`OnGet`), renders the markup and throws the object away.

Listing 9.7: nothing survives between requests unless it is stored somewhere that does — the rule of PHP, the opposite of the shared servlet fields Chapter 7 warned about. Book: Section 9.6.4.

Razor: One Character, @

Listing 9.8 The first half of `Pages/Razor.cshtml`: a C# block of variables, then explicit and implicit expressions, a method call, encoding, `@Html.Raw`, an e-mail address and `@@`.

```
@page
@{
    ViewData["Title"] = "Razor tour";
    var kilos = 5;
    var pricePerKg = 8000;
    var note = "<b>Please do not iron the batik</b>";
    string[] racks = ["A01", "A02", "A03"];
    var status = "finished";
}
<h1>Razor, one idea per line</h1>
<ol>
    <li>An expression: 2 + 3 = @(2 + 3)</li>
    <li>A variable: @kilos kg at @pricePerKg per kg = @(kilos * pricePerKg)</li>
    <li>A method call: @LaundryRules.Rupiah(kilos * pricePerKg)</li>
    <li>Text is encoded, so this stays text: @note</li>
    <li>Unless you insist (trusted markup only): @Html.Raw("<em>express</em>")</li>
    <li>An e-mail address is not code: cs@profitina-laundry.example</li>
    <li>Two at signs make one: @@profitina_laundry</li>
```

Listing 9.8: `@(2 + 3)` is explicit; `@kilos` is implicit and ends at the space, so a product needs parentheses. `@note` prints `<b>`: Razor HTML-encodes everything it prints. Book: Section 9.7.1.

Razor: Decisions, Loops, Comments

Listing 9.9 The second half of `Pages/Razor.cshtml` (its `@switch` left out): `@if` and `@foreach` choose and repeat HTML; `<text>` writes text from a code block; a Razor comment and an HTML comment.

```
    @if (kilos > 4)
    {
        <li>A decision: more than 4 kg, so a free laundry bag.</li>
    }
    else
    {
        <li>A decision: a small load.</li>
    }
    @foreach (var rack in racks)
    {
        <li>A loop: rack @rack</li>
    }
    <li>Plain text inside a code block: @{ var total = kilos * pricePerKg; <text>total =
@total</text> }</li>
</ol>
@* A Razor comment: removed on the server, never sent. *@
<!-- An HTML comment: sent to the browser, visible with View Source. -->
<p>Rendered at @DateTime.Now.ToString("HH:mm:ss") by the server.</p>
```

In Listing 9.9 the braces of `@if` and `@foreach` contain HTML again; `@switch` works the same way. `@* … *@` is removed on the server; `<!-- … -->` travels to every browser. Book: Section 9.7.1.

Razor Next to PHP and JSP

Table 9.4 Razor next to PHP and JSP. In Razor the short form is the safe form.

To …	Razor	PHP (Chapter 3)	JSP (Chapter 7)
print a value, encoded	`@x`, `@(a + b)`	`<?= htmlspecialchars($x) ?>`	`<c:out value="${x}"/>`
print markup unencoded	`@Html.Raw(x)`	`<?= $x ?>`	`${x}`
run statements	`@{ var t = a * b; }`	`<?php $t = $a * $b; ?>`	`<% int t = a * b; %>`
decide	`@if (c) { … } else { … }`	`<?php if ($c): ?> … <?php endif; ?>`	`<c:if test="${c}">…</c:if>`
repeat	`@foreach (var r in racks) { … }`	`<?php foreach ($racks as $r): ?>`	`<c:forEach var="r" items="${racks}">`
comment, not sent	`@* … *@`	`<?php /* … */ ?>`	`<%-- … --%>`

Table 9.4: in Razor the short form is the safe form. `@Html.Raw` is for markup your own program built — never for anything a visitor typed. Every page shares one frame, `_Layout.cshtml`, whose `@RenderBody()` marks the hole each page fills. Book: Sections 9.7.1 and 9.7.2.

A Page Model That Asks for What It Needs

Listing 9.10 The class in `Pages/Index.cshtml.cs` of LiveStatus: a primary constructor asks for an `ILaundryStore`; `OnGet` fills two properties for the price list.

```
// The page model: the C# half of Index.cshtml. ASP.NET Core creates one for every request
// and hands it the store it asks for in its constructor (dependency injection).
public class IndexModel(ILaundryStore store) : PageModel
{
    public IReadOnlyList<Service> Services { get; private set; } = [];
    public IReadOnlyList<Outlet> Outlets { get; private set; } = [];

    public void OnGet()                     // runs for GET /  (and GET /Index)
    {
        Services = store.Services();
        Outlets = store.Outlets();
    }
}
```

> **DEFINITION. Dependency injection** is the arrangement in which a class *declares* the objects it needs, usually as constructor parameters, and the service container creates and supplies them.

Listing 9.10 never writes `new CsvLaundryStore(...)`: it does not know which implementation it receives, so the CSV store and the SQLite store can replace each other without the page changing. Book: Section 9.8.1.

Handlers: Which Method Answers

Table 9.5 How a request finds its handler: `On`, the HTTP method, optionally a name given by `?handler=`, optionally `Async`. Parameters are filled from the query string, the form or the route by name.

Request	Handler called	In this chapter
`GET /Quote`	`OnGet()`	the empty quotation form
`POST /Quote`	`OnPost()`	the filled-in form comes back
`GET /Status?handler=Live&bill=…`	`OnGetLive(string? bill, string? phone)`	the bill's status as JSON
`POST /Counter?handler=Advance&id=B2952`	`OnPostAdvance(string id)`	move a bill one step

A handler may return `void` (render the markup) or an `IActionResult`: `Page()`, `RedirectToPage(…)`, `new JsonResult(data)`, `StatusCode(304)`.

Table 9.5: that freedom is the key to the Laundry feature — one page can answer a browser with HTML and a script with JSON. Book: Section 9.8.2.

Model Binding: the Input Class

Listing 9.11 The input class of `Pages/Quote.cshtml.cs`: validation attributes describe the form; `[Display]` supplies the label people read.

```
public class QuoteInput
{
    [Required(ErrorMessage = "Choose a service.")]
    [Display(Name = "Service")]
    public string ServiceId { get; set; } = "S0001";

    [Range(0.1, 999, ErrorMessage = "Enter an amount from 0.1 to 999.")]
    [Display(Name = "Amount (kg, items or pairs)")]
    public decimal Amount { get; set; } = 1;
}

public record QuoteResult(string Service, string Quantity, long Total, string ReadyBy);
```

Listing 9.11: binding copies the fields `Input.ServiceId` and `Input.Amount` into these properties, converting `"2.1"` to the `decimal` 2.1; validation checks `[Required]` and `[Range]` and records failures in `ModelState`. Book: Section 9.8.3.

Validation and the Price in OnPost

Listing 9.12 The first half of `OnPost` in `Pages/Quote.cshtml.cs`: two rules attributes cannot express, and the same form again when `ModelState` is not valid.

```
    public IActionResult OnPost()                    // POST /Quote: the form came back
    {
        FillList();
        var service = store.Services().FirstOrDefault(s => s.Id == Input.ServiceId);
        if (service is null)
        {
            ModelState.AddModelError("Input.ServiceId", "Choose a service from the list.");
        }
        else if (!service.Description.Contains("(per kg)") && Input.Amount !=
decimal.Truncate(Input.Amount))
        {
            ModelState.AddModelError("Input.Amount", "A whole number of items, for example 2.");
        }
        if (!ModelState.IsValid)
        {
            return Page();                           // the same form, with the messages
        }
```

Listing 9.12: if `ModelState.IsValid` is false, `Page()` returns the same form with the visitor's values and the messages. The quotation changes nothing, so `Page()` is right; a handler that changes data should end with `RedirectToPage` (Post/Redirect/Get). Otherwise `OnPost` rounds up with `Math.Ceiling` and sets `Result`. Book: Sections 9.8.3 and 9.8.5.

The Price Rule, Worked

With a the amount brought and p_s the price of service s per unit, Equation (9.1) rounds kilograms **up** and Equation (9.2) multiplies:

$$u = \lceil a \rceil \tag{9.1}$$

$$P = u \times p_s \tag{9.2}$$

> **WORKED EXAMPLE.** 2.1 kg of curtains (S0010, Rp 15.000 per kg): $u = \lceil 2.1 \rceil = 3$ and $P = 3 \times 15000 = 45000$, Rp 45.000; curtains promise 120 hours. 2.5 bed covers (S0008, per item) satisfy `[Range]`, but `OnPost` adds “A whole number of items, for example 2.”

For items and pairs a is already whole; the ceiling is `Math.Ceiling` in C#. Book: Section 9.8.3.

The Quotation Page, Drawn

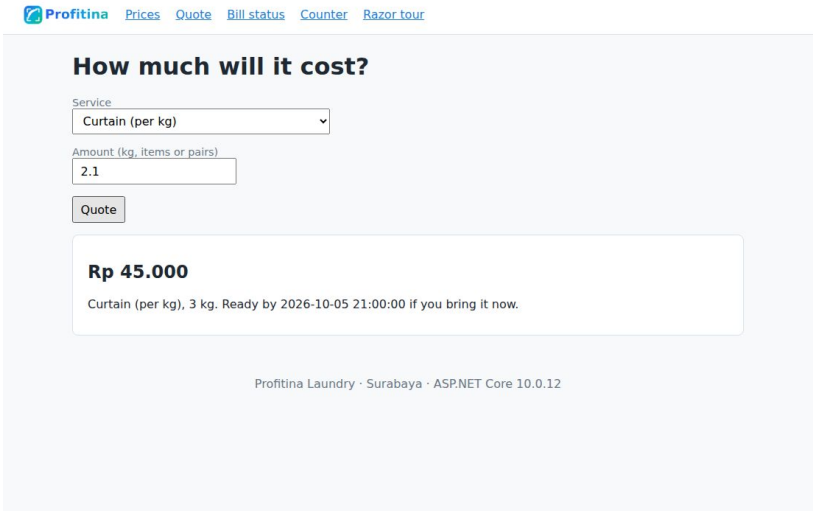


Figure 9.6 The quotation page after posting 2.1 kg of curtains, drawn by the build machine's headless Chromium: Rp 45.000 for 3 kg, ready five days later.

Figure 9.6 is exactly the worked example: the build machine's headless Chromium posted 2.1 kg of curtains and drew Rp 45.000 for 3 kg, ready five days later. Book: Section 9.8.3.

Tag Helpers and the Antiforgery Token

Listing 9.13 The form of `Pages/Quote.cshtml`: `asp-for`, `asp-items` and `asp-validation-for` are tag helpers that write names, ids, options and messages.

```
<form method="post">
    <p>
        <label asp-for="Input.ServiceId"></label>
        <select asp-for="Input.ServiceId" asp-items="Model.ServiceItems"></select>
        <span asp-validation-for="Input.ServiceId" class="error"></span>
    </p>
    <p>
        <label asp-for="Input.Amount"></label>
        <input asp-for="Input.Amount" type="number" step="0.1" min="0.1">
        <span asp-validation-for="Input.Amount" class="error"></span>
    </p>
    <button>Quote</button>
```

A **tag helper** is a C# class that runs on the server and rewrites a matching element; the form also receives a hidden `__RequestVerificationToken`.

CSRF: a page on *another* site makes the visitor's browser post to *your* site, cookies included. Razor Pages checks the token for every `POST` handler automatically.

Listing 9.13: the helpers also add a hidden `__Invariant` field, so `2.1` is never misread as 21 on a computer set to Indonesian. Chapter 7 had to write a filter for CSRF; here a post without the token receives 400. Book: Section 9.8.4.

Postback and View State

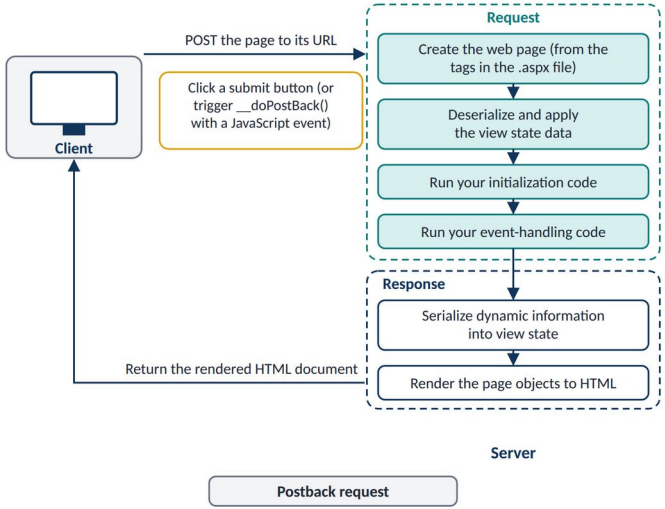


Figure 9.7 A *postback*: clicking a submit button, or a JavaScript `__doPostBack()`, posts the page to its own URL; the server recreates the page, deserializes and applies view state, runs initialization and event-handling code, serializes view state again and renders.

Figure 9.7: a button click posts the whole form back to the same page; the server recreates the page, restores every control from the hidden `__VIEWSTATE`, runs `Page_Load` with `IsPostBack == true`, calls the handler, and renders again. Book: Section 9.9.3.

The Cost of Hiding the Web

The page syntax: directives `<%@ … %>`, a `<script runat="server">` block, `<%= %>` render blocks, server controls such as `<asp:Label runat="server">`.

Productive for internal forms: `btnQuote_Click` is called for you, controls keep their values, validators such as `RequiredFieldValidator` sit in the markup.

Costly as the web grew: view state of hundreds of kilobytes per click, one form per page, HTML written by controls, every interaction a `POST`.

The UpdatePanel gave Ajax *behavior* while keeping the *model*: a `Timer` posts the entire form every 10 seconds, and the server runs the whole page lifecycle to return one panel.

> **THE KEY INSIGHT.** Web Forms tried to hide the web behind the model of a desktop form. Everything that replaced it exposes HTTP and HTML plainly and gives good tools for working *with* them. That is why a Razor Page has `OnGet` and `OnPost` instead of `btnQuote_Click`.

The modern version of the live status sends a `GET` with one header and, when nothing changed, receives `304 Not Modified` with no body. Book: Sections 9.9.2–9.9.6.

Every Web Forms Idea, in Modern Form

Table 9.6 Every Web Forms idea of this chapter in the form that replaced it. The right-hand column is what this book builds and runs.

Web Forms idea	Modern Razor Pages form	Where
`.aspx` + code-behind	`.cshtml` + page model	Section 9.6.4
`Page_Load`, `IsPostBack`	`OnGet` and `OnPost`	Section 9.8.2
`btnX_Click`	named handlers, `?handler=X`	Sections 9.8.2, 9.10.7
Server controls, view state	HTML + tag helpers; state in the request, database or session	Section 9.8.4, Chapter 14
Validation controls	DataAnnotations + `ModelState`	Section 9.8.3, Chapter 14
Master page	`_Layout.cshtml` + `@RenderBody()`	Section 9.7.2
`UpdatePanel` + `Timer`	`fetch()` + a JSON handler + `ETag`	Section 9.10
GridView + data source controls	a `@foreach` table over a list from a service	Chapter 13
ADO.NET in `App_Code`	ADO.NET in a service registered in `Program.cs`	Chapter 15
`Web.config`; IIS on Windows	`appsettings.json` + `Program.cs`; Kestrel anywhere	Sections 9.6.1–9.6.3

Table 9.6 is also the map of Chapters 11 to 15: GridView in Chapter 13, validation and state in Chapter 14, ADO.NET in Chapter 15. Book: Section 9.9.7.

The Live Bill Status: the Story

A customer brings 5 kg to the Wiyung outlet at 20:09 on 30 September 2026, pays Rp 20.000 of Rp 40.000, and leaves with bill **B2952**.

At home: *Where is my laundry?*, bill number and phone — *In process*, ready by 20:09 on 2 October, Rp 20.000 still to pay. The page stays open.

At the outlet the washer presses *Mark finished*. Within ten seconds, **without anyone touching the customer's page**, the badge says *Ready for pickup*.

The progress bar, the history and a sentence for screen readers change too. The next page shows the result, captured by the build machine's headless browser. Book: Section 9.10.1.

What the Customer Sees

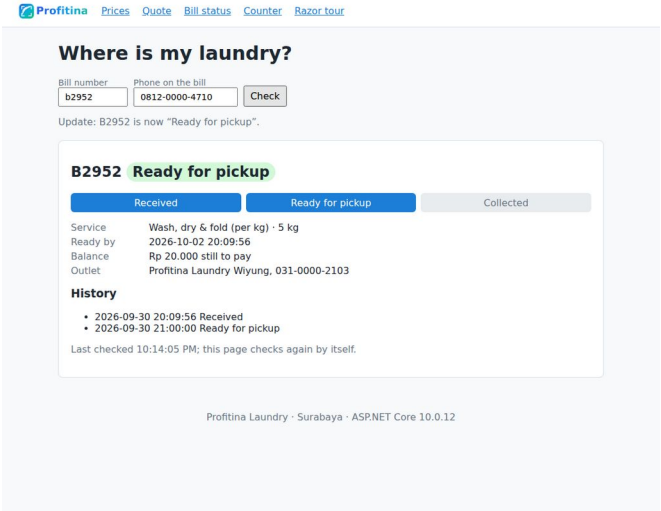


Figure 9.8 The customer's page after the counter marked B2952 finished. It was never reloaded: badge, progress bar, history and message were changed by `live-status.js`. (Razor Pages track; real screenshot.)

Figure 9.8 was never reloaded: badge, progress bar, history and message were changed by `live-status.js`. At the counter, Hand over was then refused, because Rp 20.000 is still owed. Book: Section 9.10.1.

The Rules Before the Code

Promised time, with $h(s)$ the turnaround of service s (24 hours express, 48 wash-fold, 120 curtains, …), in Equation (9.3); the balance after payments $m_1, \ldots, m_k$ in Equation (9.4):

$$t_{\text{promised}} = t_{\text{arrived}} + h(s) \tag{9.3}$$

$$\text{balance} = \text{payment} - \sum_{i=1}^{k} m_i \tag{9.4}$$

> **WORKED EXAMPLE.** B2952 arrived at 2026-09-30 20:09:56 for S0001, 5 kg at Rp 8.000: payment $= 5 \times 8000 = 40000$. With $h(\text{S0001}) = 48$ the promised time is **2026-10-02 20:09:56**. One payment of Rp 20.000: balance $= 40000 - 20000 = 20000$.

The sum is 0 when $k = 0$. All four tracks must apply exactly these rules. Book: Section 9.10.2.

A Version Label: the ETag

A bill's visible state changes only when a status event or a payment is added, so its version is its number, its count of status events e_b and its count of payments k_b:

$$v(b) = (b, e_b, k_b) \tag{9.5}$$

DEFINITION. An **ETag** (*entity tag*) is a short text the server sends in the `ETag` header to label the version of a response. A client that has that version sends it back in `If-None-Match`; if it still matches, the server answers `304 Not Modified` with no body.

Equation (9.5) is written `"b-e-k"`: `"B2952-1-1"` at the start, `"B2952-2-1"` after Mark finished. Counter rules: `queued` → `finished` at any time; `finished` → `departed` only when the balance is zero; anything else is `409 Conflict`. Book: Section 9.10.2.

The Counter's Rule, in One Lock

Listing 9.14 The heart of `Advance` in `Data/CsvLaundryStore.cs`: inside one `lock (gate)`, compute the balance and apply the two status rules.

```
            var balance = b.Payment - b.Paid;
            switch (b.Status)
            {
                case "queued":
                    b.Status = "finished";
                    b.Finished = when;
                    break;
                case "finished" when balance > 0:
                    throw new LaundryProblem(409, "conflict",
                        $"{id}: balance {LaundryRules.Rupiah(balance)} open. Record the payment
first.");
                case "finished":
                    b.Status = "departed";
                    b.Departed = when;
                    break;
                default:
                    throw new LaundryProblem(409, "conflict", $"{id} has already been collected.");
            }
```

Listing 9.14: `case "finished" when balance > 0` expresses "finished, but money is owed". The `lock` plays the part of a transaction: two cashiers pressing at once cannot both see `queued`. Pages see only the interface `ILaundryStore`. Book: Section 9.10.3.

One Minimal API in Program.cs

Listing 9.15 The end of `Program.cs` of LiveStatus: `/api/status` mapped to a lambda instead of a page, with the ETag and 304 by hand.

```
// 3. The shared JSON address of api.md, answered without a page (a "minimal API").
//    It gives exactly what /Status?handler=Live gives, so any track's front end can use it.
app.MapGet("/api/status", (string? bill, string? phone, HttpContext http, ILaundryStore store) =>
{
    try
    {
        var (etag, view) = store.LookUp(bill, phone);
        http.Response.Headers.ETag = etag;
        http.Response.Headers.CacheControl = "no-cache";
        if (http.Request.Headers.IfNoneMatch == etag)
        {
            return Results.StatusCode(StatusCodes.Status304NotModified);
        }
        return Results.Json(view, LiveJson.Options);
    }
    catch (LaundryProblem p)
    {
        return Results.Json(p.Body, LiveJson.Options, statusCode: p.Status);
    }
});
```

Listing 9.15 is a minimal API: parameters are filled by name (`bill`, `phone`) and by type (`HttpContext`, `ILaundryStore`). Earlier in the file, `AddSingleton<ILaundryStore, CsvLaundryStore>()` is the one line that chooses the store. Book: Section 9.10.4.

One Request Through the Application

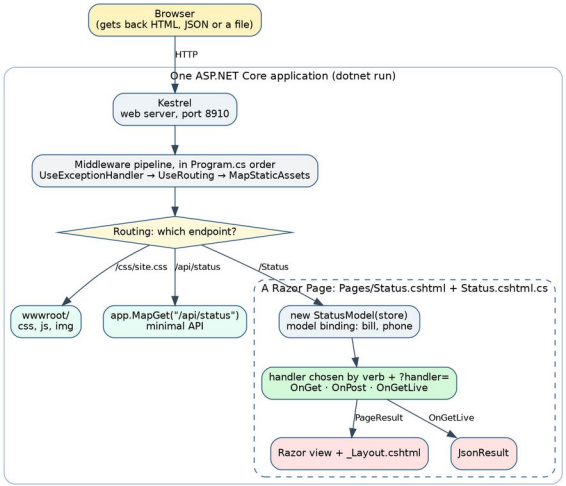


Figure 9.9 One ASP.NET Core application. Kestrel receives the request; the middleware runs in `Program.cs` order; routing chooses a static file, the minimal API or a Razor Page; for a page, a new page model is created, bound and asked to run the handler named by the method and `?handler=`, which returns a rendered page or JSON.

Figure 9.9: Kestrel receives; middleware runs in `Program.cs` order; routing chooses a static file, the minimal API or a Razor Page; a page model is created, bound and asked to run the handler named by the method and `?handler=`. Book: Section 9.10.5.

HTML for People, JSON for Scripts

Listing 9.16 `OnGetLive` in `Pages/Status.cshtml.cs`: the Ajax endpoint at `/Status?handler=Live`, with the ETag, the 304 and JSON errors.

```
    // GET /Status?handler=Live&bill=..&phone=..   the same answer as JSON, for fetch()
    public IActionResult OnGetLive(string? bill, string? phone)
    {
        try
        {
            var (etag, view) = store.LookUp(bill, phone);
            Response.Headers.ETag = etag;
            Response.Headers.CacheControl = "no-cache";
            if (Request.Headers.IfNoneMatch == etag)
            {
                return StatusCode(StatusCodes.Status304NotModified);   // nothing new: no body
            }
            return new JsonResult(view, LiveJson.Options);
        }
        catch (LaundryProblem p)
        {
            return new JsonResult(p.Body, LiveJson.Options) { StatusCode = p.Status };
        }
    }
```

Listing 9.16: a 304 has no body and no serialization at all. Beside it, `OnPost` draws the same answer on the server for a browser without JavaScript: the page is progressively enhanced. Book: Section 9.10.5.

The Ajax Engine: check()

Listing 9.17 The first half of `check` in `live-status.js`: ask with `If-None-Match`, back off, ignore a 304.

```
async function check(bill, phone) {
  timer = null;
  const headers = { Accept: "application/json" };
  if (etag) headers["If-None-Match"] = etag;          // "only if it changed, please"
  let response;
  try {
    response = await fetch(url(bill, phone), { headers, cache: "no-store" });
  } catch (e) {                                        // network down, server stopped
    delay = Math.min(delay * 2, 60000);
    say("Cannot reach the shop right now; trying again in " + delay / 1000 + " s.");
    timer = setTimeout(() => check(bill, phone), delay);
    return;
  }
  delay = every;
  $("st-checked").textContent = new Date().toLocaleTimeString();

  if (response.status === 304) {                       // same as before: nothing to draw
    timer = setTimeout(() => check(bill, phone), delay);
    return;
  }
```

Listing 9.17: `cache: "no-store"` keeps the browser cache out of the way; a network failure doubles the delay (backoff). Book: Section 9.10.6.

The Ajax Engine: the Answer

Listing 9.18 The second half of `check`: an error answer hides the card and stops; a good answer is drawn, and the next check is scheduled unless the bill has been collected.

```
    const data = await response.json();
    if (!response.ok) {                                  // 404, 400, 429: the server explains
      $("result").hidden = true;
      etag = null;
      shown = null;
      say(data.error ? data.error.message : "Something went wrong.");
      return;                                            // do not keep asking about a wrong bill
    }
    etag = response.headers.get("ETag");
    draw(data);
    if (finished(data)) {
      say(data.bill_id + " has been collected. Thank you!");
      return;
    }
    timer = setTimeout(() => check(bill, phone), delay);
  }
```

Listing 9.18: a wrong bill or phone is not asked about again. `draw` puts every value in with `textContent`, so text stays text. Book: Section 9.10.6.

Submitting Without Reloading

Listing 9.19 The submit listener of `live-status.js`: `event.preventDefault()` turns an ordinary form into an Ajax form.

```
form.addEventListener("submit", (event) => {
  event.preventDefault();                              // no page reload: this is the RIA part
  if (timer) clearTimeout(timer);
  etag = null;
  shown = null;
  const bill = form.elements.bill.value.trim();
  const phone = form.elements.phone.value.trim();
  say("Checking " + bill.toUpperCase() + "…");
  history.replaceState(null, "", "?bill=" + encodeURIComponent(bill));
  check(bill, phone);
});
```

The address bar receives `?bill=…` but **not the phone**: addresses end up in history, bookmarks and screenshots.

Every value is drawn with **`textContent`**, never `innerHTML`; the message line is an `aria-live` region; a hidden tab stops asking.

Without the first line of Listing 9.19 the browser would post and load a new page. The same script, byte for byte, serves all four tracks. Book: Section 9.10.6.

The Whole Conversation

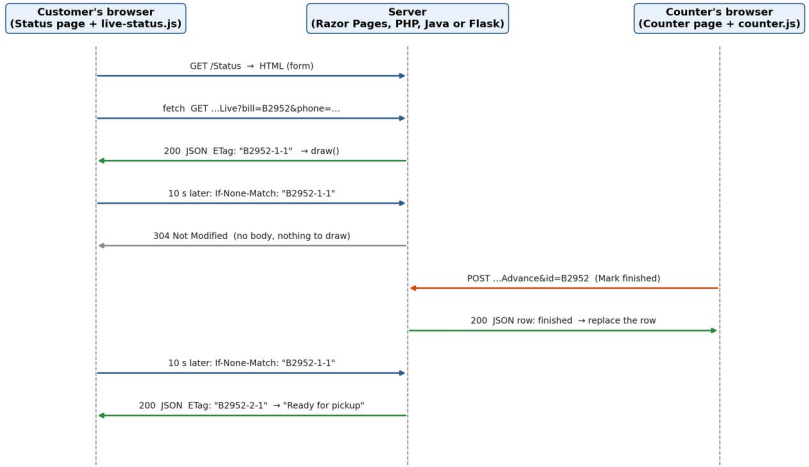


Figure 9.10 The live status as a sequence diagram: the first check draws the bill; ten seconds later a check with `If-None-Match` receives an empty `304`; the counter posts *Mark finished*; the next check receives `200` with a new ETag, and the page changes.

Figure 9.10: the first check draws the bill; ten seconds later an empty 304; the counter posts Mark finished; the next check receives 200 with a new ETag, and the page changes. Book: Section 9.10.6.

Watching It Work

Listing 9.20 The real transcript of the end-to-end browser test on the Razor Pages track: responses 200, 304 and 200, and the customer's page changed 9.0 seconds after the counter's click.

```
customer sees: B2952 - In process
address bar  : http://127.0.0.1:8910/Status?bill=b2952
responses so far: [200, 304]
counter says : B2952 is now “Ready for pickup”.
counter says : B2952: balance Rp 20.000 open. Record the payment first.
customer sees: Ready for pickup after 9.0 s, without reloading
announcement : Update: B2952 is now “Ready for pickup”.
history      : 2026-09-30 20:09:56 Received | 2026-09-30 21:00:00 Ready for pickup
responses    : [200, 304, 200]
```

Listing 9.20: responses 200, 304, 200; the customer's page changed 9.0 seconds after the click. The counter's script sends the antiforgery token in the `RequestVerificationToken` header, and disables its button while waiting. Book: Sections 9.10.7 and 9.10.8.

Polling: What It Costs

With N pages open, each asking every T seconds, the server receives Equation (9.6) requests per hour:

$$R = \frac{3600\,N}{T} \tag{9.6}$$

> **WORKED EXAMPLE.** 300 customers with $T = 10$ s: $R = 3600 \times 300/10 = 108000$ requests per hour, $108000/3600 = 30$ per second, almost all bodiless 304s. With $T = 2$ s the load is five times larger, to learn of changes that happen a few times in a bill's life.

The hidden-tab pause keeps N down to the pages people actually look at. Book: Section 9.10.9.

From Asking to Being Told

Table 9.7 From asking to being told. All four are available to every track of this chapter.

Technique	How it works	Cost and complexity	Typical use
Polling (this chapter)	The browser asks every T seconds; ETags make unchanged answers tiny	Simplest; delay up to T; load $\propto N/T$ (Equation (9.6))	Status pages, dashboards
Long polling	The server holds each request open until something changes	Near-instant; one open request per page	Chat on older infrastructure
Server-Sent Events (SSE)	One long-lived HTTP response in which the server writes events; `EventSource` in the browser	Near-instant, one way, plain HTTP	Feeds, notifications
WebSockets (SignalR in .NET)	A two-way connection upgraded from HTTP	Instant, both ways; the server keeps connections	Chat, collaboration, games; Blazor Server runs on SignalR

Table 9.7: for the Laundry, polling every ten seconds is right — the change happens once or twice in a bill's life, and it works through every proxy and on every phone. Book: Section 9.10.9.

The Same Address in PHP

Listing 9.21 The status branch of `php/router.php`: look up, set `ETag` and `Cache-Control`, compare with `If-None-Match`, answer 304 or the JSON.

```
    if ($method === 'GET' && $path === '/api/status') {
        [$etag, $data] = $laundry->lookUp($_GET['bill'] ?? null, $_GET['phone'] ?? null);
        header('ETag: ' . $etag);
        header('Cache-Control: no-cache');
        if (($_SERVER['HTTP_IF_NONE_MATCH'] ?? '') === $etag) {
            http_response_code(304);                // nothing new: no body at all
            return true;
        }
        send_json(200, $data);
```

Listing 9.21 is `OnGetLive` again. PHP's built-in server runs `router.php` for every request; it answers `/api/…` and returns `false` for anything else, so the static files are sent as they are. Book: Section 9.11.

The Same Address in Python

Listing 9.22 The status view of `python/app.py`: the same five steps in Flask.

```python
@app.get("/api/status")
def status():
    etag, data = ls.look_up(db(), request.args.get("bill"), request.args.get("phone"))
    if request.headers.get("If-None-Match") == etag:
        return "", 304, {"ETag": etag}                  # nothing new: no body at all
    response = jsonify(data)
    response.headers["ETag"] = etag
    response.headers["Cache-Control"] = "no-cache"
    return response
```

The PHP, Java and Flask tracks demand `application/json` for the advance request, or answer **415**: a plain form cannot send JSON across sites — their CSRF defense.

Listing 9.22: route, look up, version, compare, answer. The Java track does the same by hand with `sendResponseHeaders(304, -1)`, where `-1` means no body. Book: Sections 9.12 and 9.13.

One Contract, Four Servers

Table 9.8 The live bill status in four languages. The vocabulary differs; the jobs — route, look up, version, compare, answer — are the same in every column.

Job	Razor Pages (C#)	PHP 8	Java (JDK HttpServer)	Python (Flask)
Start, port	`dotnet run`, 8910	`php -S … router.php`, 8920	`java … LiveStatusServer`, 8930	`python app.py`, 8940
Status address	`/Status?handler=Live`, `/api/status`	`/api/status`	`/api/status`	`/api/status`
Route to code	file under `Pages/` + handler; `MapGet`	`router.php`	`createContext`	`@app.get`, `@app.post`
JSON out	`JsonResult`, `Results.Json`	`json_encode`	hand-written `Json.write`	`jsonify`
Data	CSV in memory (SQLite with one package)	PDO + SQLite	JDBC + SQLite	`sqlite3`
CSRF defense	antiforgery token, automatic	JSON required (415)	JSON required (415)	JSON required (415)
Front end	`Status.cshtml` + `live-status.js`	`status.html` + `live-status.js`	the same	the same

Table 9.8: `tests/check_tracks.py` asked each server the same questions and compared the parsed JSON — identical. Once the browser asks for data, front end and back end are joined only by a contract: addresses, status codes and the shape of the JSON. Book: Section 9.14.

Profitina in Practice

Data, not pages: the dashboard of `app.profitina.com` is drawn from one JSON call, `GET /api/me/dashboard` — the middle column of Table 9.3.

Push where polling would be wrong: the community feed uses Server-Sent Events.

Optimistic updates, escaping without exception, data minimization under UU PDP No. 27/2022.

Profitina is Python on FastAPI and PostgreSQL, not .NET; the ideas carry because they are about the browser and HTTP. The runnable case study is always Profitina Laundry. Book: Section 9.15.

Common Mistakes

Setup: installing the runtime instead of the SDK; mixing up the .NET Framework and .NET; running a public server in Development.

Razor Pages: forgetting `@page`; misnaming a handler; posting from a script without the antiforgery token; parsing numbers with the machine's culture.

Safety: `@Html.Raw` or `innerHTML` with data you did not write; different answers for "no such bill" and "wrong phone"; personal data in the address bar.

Ajax: forgetting `event.preventDefault()`; polling too often or forever; an ETag that does not change when the data does; letting the browser cache hide the 304.

Each mistake is listed in the book with its cure. Book: Section 9.16.

Summary: Five Things to Remember

1. The plugins are gone; HTML, CSS, JavaScript and HTTP remain. Ajax asks for **data** and changes part of the page.

2. .NET 10 is the LTS until November 2028; ASP.NET Core runs anywhere on Kestrel; Web Forms is history with modern heirs.

3. A Razor Page is `@page` markup plus a page model with handlers; `Program.cs` registers services, then builds the pipeline.

4. Razor encodes what it prints; binding, validation and tag helpers make forms; the antiforgery token is automatic.

5. ETags make “nothing new” cheap; polling costs Equation (9.6); one JSON contract lets four servers share one front end.

Book: Section 9.17.

Next Time

Chapter 10 — Three Systems Demonstrated: PHP, JSP, ASP.NET. One small application built three ways, with logins, sessions and the Post/Redirect/Get pattern.

Before then: work Exercises 9.1, 9.2, 9.4 and 9.5 by hand, then check each answer against a running track.

The solutions to all twelve exercises are in the book. Book: Sections 9.18 and 9.19.

Lecture 10 — Three Systems Demonstrated: PHP, JSP, and ASP.NET

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 10

Lecturer: Irfan Subakti — yifana@gmail.com

Today one small piece of Profitina Laundry is built five times, and we learn to read any one of the five and point at its twin in the other four. Book: Chapter 10, Sections 10.1–10.16.

What This Meeting Covers

1. One contract, five builds — the feature set, the shop's rules, and the four layers every build shares.

2. The servers — XAMPP, Tomcat, IIS and the .NET SDK on Windows, and their cross-platform equivalents.

3. Shared parts — the rules in one file per language; one password-hash format verified in four languages.

4. The five builds — PHP 8 with PDO, Jakarta Servlets and JSP, the JDK HttpServer, ASP.NET Core Razor Pages, Flask.

5. One acceptance test — eighteen checks from the outside; where the builds differ on the wire.

6. Side by side, deployment, and which to choose.

Where all five do the same thing, that thing is the web; where they differ, the difference is a design decision. Book: Section 10.1.

Where We Left Off: Web or System?

Chapters 3 and 5 taught **PHP** with MySQL through PDO; Chapter 7 **Java** servlets and JSP in a container; Chapter 9 **ASP.NET Core Razor Pages** on .NET 10.

One system at a time is the right way to learn a system — and the wrong way to understand the web.

Is it the web that makes you escape text, or PHP? Does the web need a session cookie, or does Tomcat? Is a “route” an idea, or a Flask decorator?

Today answers by **experiment**: one feature set, five builds, one test.

Four of the five builds ran and were tested over real HTTP; the servlet build was compiled and packaged but not deployed, because Tomcat could not be downloaded in the build environment. Book: Section 10.1.

The Contract

Table 10.1 The feature set, first part: the login, the customer list and the new-bill form.

Method and address	Who may use it	What happens	Answer
`GET /login`	anyone	the login form, with a hidden anti-forgery token	200
`POST /login`	anyone	check e-mail and password against the stored PBKDF2 hash	redirect to `/customers`, or 401 with "Wrong e-mail or password."
`POST /logout`	logged in	forget the login	redirect to `/login`
`GET /customers?q=…`	logged in	customers whose name or phone contains `q`; the count, and the first 20	200
`GET /bills/new`	logged in, not a washer	the new-bill form	200, or 403 for a washer

Table 10.1 says nothing about PHP, Java, C# or Python; it says what a browser sends and what it gets back. That is all a browser can ever know about a server, and it is why one test can check five programs. Book: Section 10.2.1.

The Contract, Continued

Table 10.2 The feature set, second part: the bill page and the two rules that apply to every address.

Method and address	Who may use it	What happens	Answer
`POST /bills/new`	logged in, not a washer	check the form; if valid, insert the bill and its first status-log row in one transaction	redirect to `/bills/<id>`, or 422 with the messages
`GET /bills/<id>`	logged in	the bill, its customer, service, charge, outlet and cashier	200, or 404
any `POST` without the right token	—	refused before any page code runs	403 (or 400, Section 10.10)
any page but `/login`, not logged in	—	sent to the login page	redirect to `/login`

Table 10.2: a refused request never reaches page code. Which redirect status each system sends — 302 or 303 — is itself one of the findings. Book: Section 10.2.1.

Three Rules of the Shop

Roles: no supervisor → **owner**; named as someone's supervisor → **supervisor**; otherwise **cashier** (section `transaction`) or **washer** (section `wash`). Washers do not take new bills.

A valid bill: a customer like `C0001` that exists; one of the twelve services; units a whole number from 1 to 100; a rack from `A01` to `C20`.

The charge, copied into the bill so that a later price change never rewrites history:

$$\text{charged} = \text{price per unit} \times \text{units} \tag{10.1}$$

> **WORKED EXAMPLE.** Three kilograms of express wash-and-fold (`S0002`, Rp 12.000 per kilogram): $12000 \times 3 = 36000$, written Rp 36.000. That is the bill every build creates in its test.

Equation (10.1): input is trimmed and upper-cased where that makes sense, so `c0001` and `a05` are accepted as `C0001` and `A05`. Book: Section 10.2.1.

Five Builds

Table 10.3 The five builds of this chapter. The two Java builds share one package of model, database and rule classes; the others are self-contained.

Folder	System	Port	Started with	Ran in this book's build
`php/`	PHP 8.4 + PDO, templates in plain PHP	9001	`php -S 127.0.0.1:9001 -t public public/index.php`	yes
`java/servlet/`	Jakarta Servlet 6 + JSP 3.1 + JSTL, a WAR for Tomcat 11	8080 (Tomcat's)	`./build.sh`, then deploy `dist/ROOT.war`	compiled and packaged; not deployed
`java/httpserver/`	the JDK's `com.sun.net.httpserver` + JDBC	9002	`./run.sh`	yes
`aspnet/`	ASP.NET Core Razor Pages, .NET 10	9003	`dotnet run`	yes
`python/`	Flask 3 + Jinja2 + `sqlite3`	9004 (9005 under gunicorn)	`python app.py`	yes, and under gunicorn

Table 10.3: all read the same `Chapter10/data/laundry.db`, except the ASP.NET build (Section 10.8). The two Java builds share one package of model, database and rule classes. Book: Section 10.2.2.

One Bill, Four Systems

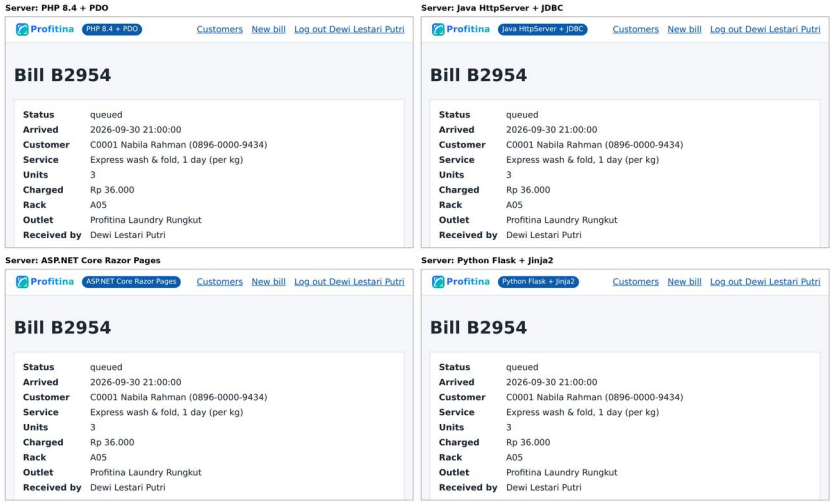


Figure 10.1 Bill B2954 as four different systems drew it: PHP (top left), the JDK HttpServer (top right), ASP.NET Core Razor Pages (bottom left) and Flask (bottom right). Same database, same form input, same HTML and stylesheet; only the badge differs.

Figure 10.1: the same login, the same form, the same database — and the same page. Only the badge in the header tells the systems apart. Book: Section 10.2.2.

Four Layers in Every Build

Rules — the shop's decisions, no web and no SQL: `rules.php`, `Rules.java`, `Rules.cs`, `rules.py`.

Store — every SQL statement, each with bound parameters, in one class or module.

Pages — the code that runs for one address: PHP functions, servlets, `PageModel` classes, Flask views.

Views — HTML with holes: PHP templates, JSP, Java text blocks, `.cshtml`, Jinja2.

> **THE KEY INSIGHT.** Put the rules where no framework can reach them. Four spellings of one set of decisions, none importing anything from a web framework: that is what lets you change frameworks without changing the shop's rules.

This is the Model–View–Controller arrangement of Chapter 7. Underneath, the schema's `CHECK` constraints and foreign keys are defense in depth. Book: Sections 10.2.3 and 10.4.1.

Three Servers on One Windows Machine

XAMPP: Apache, MySQL (MariaDB) and PHP in one installer, with a control panel and phpMyAdmin; `htdocs` is the document root.

Tomcat inside XAMPP is from the `javax` era — 7.0 in these screens, 8.5.96 in today's XAMPP 8.2.12: a `jakarta` WAR answers 404 there. This book uses Tomcat 11.

IIS is part of Windows, switched on under *Turn Windows features on or off*; the **.NET SDK** gives `dotnet new` and `dotnet run`; Visual Studio is Windows-only.

Without Windows: `php -S`, Tomcat 11 as a zip, Kestrel everywhere, Flask wherever Python runs.

A port is a number the operating system uses to deliver traffic to the right program; two programs cannot listen on the same one. Book: Sections 10.3.1–10.3.7.

The Tomcat That XAMPP Carried

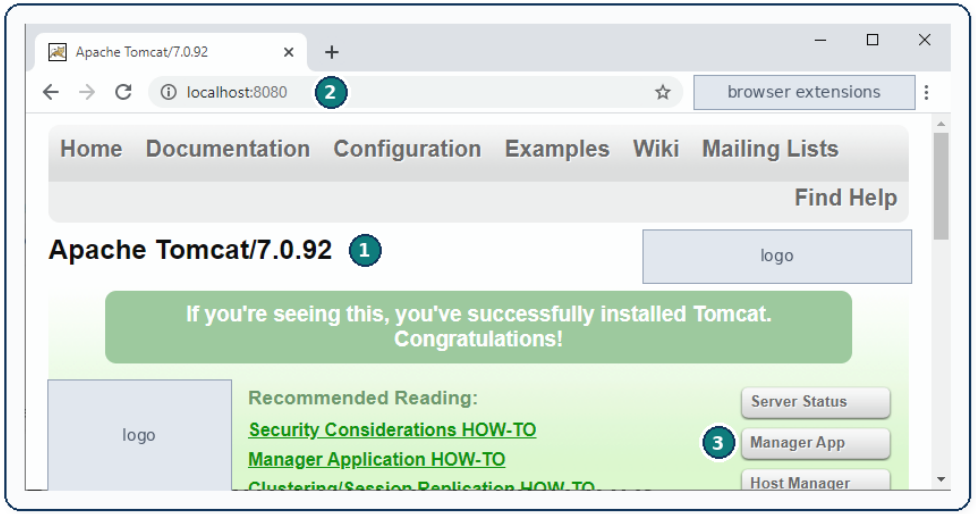


Figure 10.2 The Tomcat that an older XAMPP carried, version 7.0.92 (1), answering on port 8080 (2), with its three management buttons (3). Its version number decides which Java packages your servlets must import.

Figure 10.2: version 7.0.92. The version number decides which Java packages your servlets must import — `javax` here, `jakarta` from Tomcat 10 on. Book: Section 10.3.2.

The Rules in Python: Patterns and Roles

Listing 10.1 The first part of `python/rules.py`: three patterns and the role of an employee, worked out from the `Employee` row.

```python
"""The shop's rules, with no web and no SQL in them: the same rules in all five versions."""
import re

RACK = re.compile(r"[ABC](0[1-9]|1[0-9]|20)")      # A01 ... C20
CUSTOMER_ID = re.compile(r"C[0-9]{4}")
UNITS = re.compile(r"[0-9]{1,3}")                    # ASCII digits only, at most 3 of them


def role(employee):
    """Owner, supervisor, cashier or washer, worked out from the Employee row."""
    if employee["spv_id"] is None:
        return "owner"
    if employee["supervises"]:
        return "supervisor"
    return "cashier" if employee["section"] == "transaction" else "washer"
```

Listing 10.1 imports nothing from Flask and runs no SQL. `php/src/rules.php`, `Rules.java` and `Rules.cs` are line-for-line translations. Book: Section 10.4.2.

The Rules in Python: a Valid Bill

Listing 10.2 `validate_bill` in `python/rules.py`: clean every field, then check it; return the cleaned values with the errors.

```
def validate_bill(form, store):
    """Return (values, errors). An empty errors dict means the bill may be stored."""
    values = {
        "customer": form.get("customer", "").strip().upper(),
        "service": form.get("service", "").strip(),
        "units": form.get("units", "").strip(),
        "rack": form.get("rack", "").strip().upper(),
    }
    errors = {}
    if not CUSTOMER_ID.fullmatch(values["customer"]):
        errors["customer"] = "A customer id looks like C0001."
    elif not store.customer_exists(values["customer"]):
        errors["customer"] = "No customer with id %s." % values["customer"]
    if values["service"] not in {s["id"] for s in store.services()}:
        errors["service"] = "Choose a service from the list."
    if not (UNITS.fullmatch(values["units"]) and 1 <= int(values["units"]) <= 100):
        errors["units"] = "Units must be a whole number from 1 to 100."
    if not RACK.fullmatch(values["rack"]):
        errors["rack"] = "A rack looks like A01 (rows A to C, 01 to 20)."
    return values, errors
```

Listing 10.2 returns the cleaned values too, so the form can be shown again with what was typed, and the store only ever sees clean values. Book: Section 10.4.2.

Four Details, Four Bugs Avoided

`fullmatch`, not `match`: `C0001; DROP TABLE` begins with a valid id. PHP needs `^…$` and the `D` modifier, or `A05\n` passes.

`[0-9]`, not `\d`: in Python 3 `\d` matches Arabic-Indic and Devanagari digits too, and `int()` converts them.

Test the text, then convert: `int("0.5")` raises, `int(" 3 ")` does not, `float("1e2")` is a hundred.

Return cleaned values, and store only those — never the raw form.

Each of these is a bug in somebody's real program. Book: Section 10.4.2.

Passwords: Why Iterations Matter

Stored as `pbkdf2_sha256$600000$<salt>$<hash>`: algorithm, iterations, sixteen random bytes of salt, 32 bytes of result, both in Base64.

A machine computing R single hashes per second against n iterations tests about Equation (10.2) guesses per second:

$$G = \frac{R}{n} \tag{10.2}$$

WORKED EXAMPLE. With $R = 10^{10}$ and $n = 600000$: $G \approx 16667$ guesses per second, so 10^9 likely passwords take about 16.7 hours per employee, and — the salts being different — about 233 hours for all fourteen. One unsalted SHA-256 would take a tenth of a second for all of them.

One login took 0.16 to 0.24 seconds in the four running builds, most of it this computation — a price nobody at the counter notices. Book: Section 10.4.3.

One Hash Format in PHP

Listing 10.3 `verify_password` in PHP: `hash_pbkdf2` with `true` for raw bytes, and `hash_equals` for the comparison.

```
/** PBKDF2 hashes in the Profitina Laundry format: pbkdf2_sha256$iterations$salt$hash */
function verify_password(string $password, string $stored): bool
{
    $parts = explode('$', $stored);
    if (count($parts) !== 4 || $parts[0] !== 'pbkdf2_sha256') {
        return false;
    }
    $salt = base64_decode($parts[2], true);
    $expected = base64_decode($parts[3], true);
    if ($salt === false || $expected === false) {
        return false;
    }
    $key = hash_pbkdf2('sha256', $password, $salt, (int) $parts[1], strlen($expected), true);
    return hash_equals($expected, $key);                              // constant-time comparison
}
```

Listing 10.3: `true` *asks* `hash_pbkdf2` *for raw bytes;* `hash_equals` *compares in constant time. The Java version uses* `PBKDF2WithHmacSHA256` *and* `MessageDigest.isEqual`*, the C# one* `Rfc2898DeriveBytes.Pbkdf2` *and* `FixedTimeEquals`*. Book: Section 10.4.3.*

The Same Format in Python

Listing 10.4 `verify_password` in Python: `hashlib.pbkdf2_hmac` from the standard library.

```python
def verify_password(password: str, stored: str) -> bool:
    try:
        algorithm, iterations, salt_b64, hash_b64 = stored.split("$")
        if algorithm != "pbkdf2_sha256":
            return False
        salt = base64.b64decode(salt_b64)
        expected = base64.b64decode(hash_b64)
        key = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt,
                                  int(iterations), dklen=len(expected))
        return hmac.compare_digest(key, expected)      # constant-time comparison
    except (ValueError, AttributeError):
        return False
```

WARNING. No build ever stores or compares a plain password, and none says *which* field was wrong: “Wrong e-mail or password.” for both, or a stranger learns which addresses belong to the staff.

Listing 10.4: an ordinary `==` stops at the first differing byte and leaks timing; the constant-time functions look at every byte. A malformed hash answers `false`, never an exception. Book: Section 10.4.3.

Six Stages, Five Builds

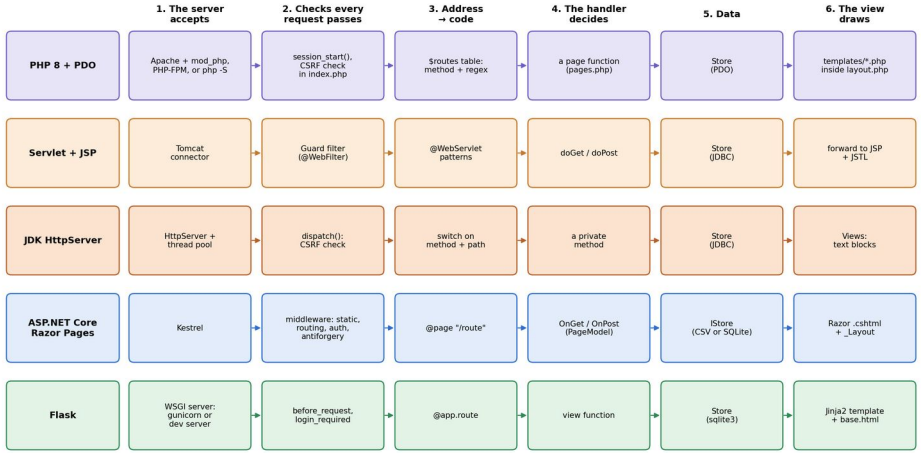


Figure 10.3 The life of one request in each of the five builds. The columns are the same six stages in every row; only the names in the boxes change.

Figure 10.3 is the map for the rest of the meeting: a server accepts, cross-cutting checks run, the address becomes code, the code decides, data is read or written, a view draws the answer. Only the names in the boxes change. Book: Section 10.5.

PHP: Nothing Survives

Shared-nothing: every request starts `public/index.php` from its first line with empty variables; at the end everything is thrown away.

Forgiving — a corrupted variable cannot affect the next visitor — but anything that must survive goes outside the program: the session file, the database.

One door: a *front controller*; only `public/` is visible to the web server, so source, templates and configuration cannot be requested.

File-per-address PHP has three weaknesses: every file must remember the checks, `.php` leaks into the URL, and every file can be requested directly. Book: Sections 10.5.1 and 10.5.2.

The Front Controller: Session and Token

Listing 10.5 The first half of `php/public/index.php`: real files handed back to the built-in server, the source loaded, the session started, and every POST checked for its token.

```
if (PHP_SAPI === 'cli-server'
        && is_file(__DIR__ . parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH))) {
    return false;                          // style.css, the logo: let the server send the file
}

require __DIR__ . '/../src/Store.php';
require __DIR__ . '/../src/rules.php';
require __DIR__ . '/../src/web.php';
require __DIR__ . '/../src/pages.php';
$config = require __DIR__ . '/../config.php';

session_set_cookie_params(['httponly' => true, 'samesite' => 'Lax']);
session_start();

$method = $_SERVER['REQUEST_METHOD'];
if ($method === 'POST' && !hash_equals(csrf_token(), (string) ($_POST['csrf'] ?? ''))) {
    abort(403, 'The form has expired. Go back, reload the page and try again.');
}
```

Listing 10.5: the session cookie is `HttpOnly` (invisible to page scripts) and `SameSite=Lax`. Because the CSRF check sits here, before any page code, no page can forget it. Book: Section 10.5.2.

The Front Controller: the Route Table

Listing 10.6 The second half of `php/public/index.php`: method, pattern and function per row; the first match wins, and captured parts become arguments.

```
// The route table: method, address pattern, function. The first match wins.
$routes = [
    ['GET',  '#^/$#',                  fn() => redirect('/customers')],
    ['GET',  '#^/login$#',             'login_form'],
    ['POST', '#^/login$#',             'login'],
    ['POST', '#^/logout$#',            'logout'],
    ['GET',  '#^/customers$#',         'customers'],
    ['GET',  '#^/bills/new$#',         'new_bill_form'],
    ['POST', '#^/bills/new$#',         'create_bill'],
    ['GET',  '#^/bills/([A-Za-z][0-9]{4})$#', 'show_bill'],
];

$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
foreach ($routes as [$verb, $pattern, $handler]) {
    if ($verb === $method && preg_match($pattern, $path, $m)) {
        $handler(...array_slice($m, 1));           // captured parts become arguments
    }
}
abort(404, 'There is no page at this address.');
```

In Listing 10.6 order is meaning: specific routes such as `/bills/new` come before patterns. Under Apache, `.htaccess` sends every address that is not a real file to `index.php`. Book: Section 10.5.2.

A PHP Page: the Login

Listing 10.7 `login` in `php/src/pages.php`: a neutral 401 with the form, or a fresh session id and a redirect.

```
function login(): never
{
    $email = strtolower(trim($_POST['email'] ?? ''));
    $employee = store()->employeeByEmail($email);
    if ($employee === null || !$employee['password_hash']
            || !verify_password($_POST['password'] ?? '', $employee['password_hash'])) {
        render('login', ['title' => 'Staff login', 'email' => $email,
                         'error' => 'Wrong e-mail or password.'], 401);
    }
    session_regenerate_id(true);                        // a fresh session id for the new login
    $_SESSION = ['user' => ['id' => $employee['id'], 'name' => $employee['name'],
                            'role' => role($employee), 'outlet' => $employee['Outlet_id']]];
    redirect('/customers');
}
```

Listing 10.7: the e-mail is trimmed and lower-cased; `session_regenerate_id(true)` defeats session fixation. Every page function is declared `: never` — it always ends the request. Book: Section 10.5.3.

A PHP Page: Post/Redirect/Get

Listing 10.8 `create_bill` in `php/src/pages.php`: 422 with the form and the messages, or store the bill and redirect to its page.

```
function create_bill(): never
{
    $user = require_login();
    may_create_bills($user);
    [$values, $errors] = validate_bill($_POST, store());
    if ($errors) {
        render('bill_new', ['title' => 'New bill', 'services' => store()->services(),
                            'values' => $values, 'errors' => $errors], 422);
    }
    $id = store()->createBill($values['customer'], $values['service'], (int) $values['units'],
                              $values['rack'], $user['id'], $user['outlet'], now());
    redirect("/bills/$id");
}
```

Listing 10.8: after a valid POST the browser's next request is a harmless `GET`, so Reload never creates a second bill. `redirect` sends 303 See Other. Book: Section 10.5.3.

Escaping by Hand

Listing 10.9 `php/templates/customers.php`. Every value that came from the database or from the visitor passes through `e()`; the count is an integer that PHP itself produced, and is printed as it is.

```
<h1>Customers</h1>
<form method="get" action="/customers">
  <label for="q">Name or phone contains</label>
  <input id="q" name="q" value="<?= e($q) ?>">
  <button>Search</button>
</form>
<p class="count"><?= $total ?> customer<?= $total === 1 ? '' : 's' ?> found<?php
  if ($total > count($rows)): ?>, showing the first <?= count($rows) ?><?php endif; ?>.</p>
<table>
  <tr><th>Id</th><th>Name</th><th>Phone</th><th>E-mail</th></tr>
  <?php foreach ($rows as $c): ?>
  <tr><td><?= e($c['id']) ?></td><td><?= e($c['name']) ?></td><td><?= e($c['phone']) ?></td><td><?=
e($c['email']) ?></td></tr>
  <?php endforeach; ?>
</table>
```

Listing 10.9: escaping is opt-in. `<?= $c['name'] ?>` would print a name unescaped and nothing would warn you. `e()` is `htmlspecialchars` with `ENT_QUOTES` and `ENT_SUBSTITUTE`. Book: Section 10.5.4.

PDO: Placeholders and LIKE

Listing 10.10 `Store::searchCustomers`: the visitor's text becomes a `LIKE` pattern whose own wildcard characters are escaped, and both statements are prepared with `?` placeholders.

```
    /** @return array{0:int,1:array} the number of matches and the first PAGE of them */
    public function searchCustomers(string $text): array
    {
        $pattern = '%' . strtr($text, ['!' => '!!', '%' => '!%', '_' => '!_']) . '%';
        $where = " WHERE name LIKE ? ESCAPE '!' OR phone LIKE ? ESCAPE '!'";
        $count = $this->db->prepare('SELECT COUNT(*) FROM Customer' . $where);
        $count->execute([$pattern, $pattern]);
        $rows = $this->db->prepare('SELECT id, name, phone, email FROM Customer' . $where
                                   . ' ORDER BY name, id LIMIT ' . self::PAGE);
        $rows->execute([$pattern, $pattern]);
        return [(int) $count->fetchColumn(), $rows->fetchAll()];
    }
```

Listing 10.10: placeholders keep `%' OR 1=1 --` as text; the `!` escapes make `%` and `_` literal, or a search for `_` would match everyone. The only concatenated value is the constant `PAGE`. Book: Section 10.5.6.

Servlets: One Object, Many Threads

Tomcat creates **one** object per servlet and filter and calls it from many threads; a listener builds the one shared `Store` at start-up.

Good: expensive things are built once. Dangerous: a field is shared by every visitor — **never put request data in a field**.

Mappings: **exact** (`/bills/new`), **path prefix** (`/bills/*`, rest in `getPathInfo()`), **empty string** (the root). Exact beats prefix, whatever the load order.

In PHP the order of the rows decided; here the kind of pattern decides. Book: Sections 10.6.1 and 10.6.2.

The Filter: One Guard for Every Request

Listing 10.11 The end of `doFilter` in `Guard`: the token checked on every POST, and anyone without a login sent to `/login`.

```
        if ("POST".equals(req.getMethod())) {
            String sent = req.getParameter("csrf") == null ? "" : req.getParameter("csrf");
            if (!MessageDigest.isEqual(sent.getBytes(StandardCharsets.UTF_8),
                    session.getAttribute("csrf").toString().getBytes(StandardCharsets.UTF_8))) {
                Pages.error(req, resp, 403, "The form has expired. Go back, reload the page and try
again.");
                return;
            }
        }
        if (!path.equals("/login") && session.getAttribute("user") == null) {
            resp.sendRedirect(req.getContextPath() + "/login");
            return;
        }
        chain.doFilter(req, resp);
```

Listing 10.11: earlier lines set UTF-8 and create a token per session; `setCharacterEncoding` must come before the first `getParameter`; `chain.doFilter` lets the request continue — a filter that returns without it has answered itself. Book: Section 10.6.3.

LoginServlet.doPost

Listing 10.12 From `doPost` of `LoginServlet`: the same neutral 401, then `changeSessionId` before the user is stored.

```
        if (e == null || e.passwordHash() == null || !Passwords.verify(password, e.passwordHash())) {
            req.setAttribute("title", "Staff login");
            req.setAttribute("email", email);
            req.setAttribute("error", "Wrong e-mail or password.");
            Pages.view(req, resp, 401, "login");
            return;
        }
        req.changeSessionId();                              // a fresh session id for the new login
        req.getSession().setAttribute("user", new User(e.id(), e.name(), Rules.role(e), e.outletId()));
        Pages.seeOther(req, resp, "/customers");
```

Listing 10.12: `changeSessionId` is the servlet API's `session_regenerate_id`. `Pages.view` forwards to a JSP under `WEB-INF`, which no URL can reach directly. Book: Section 10.6.4.

Without a Container: the HttpServer Build

Listing 10.13 The route table in `dispatch` of `Main.java`: a `switch` on method and path, and a regular expression for `/bills/<id>` in the `default` branch.

```
            Matcher bill = BILL.matcher(path);
            switch (method + " " + path) {
                case "GET /" -> w.redirect("/customers");
                case "GET /login" -> page(w, 200, "Staff login", Views.login(csrf(w), "", null));
                case "POST /login" -> login(w);
                case "POST /logout" -> logout(w);
                case "GET /customers" -> customers(w);
                case "GET /bills/new", "POST /bills/new" -> newBill(w);
                default -> {
                    if (method.equals("GET") && bill.matches()) {
                        showBill(w, bill.group(1));
                    } else {
                        error(w, 404, "There is no page at this address.");
                    }
                }
            }
```

Listing 10.13: `dispatch` is the container and the filter in one method — static files, the CSRF check, then the switch (exact match) and the regex (prefix). Sessions are written out by hand: an unguessable id, a `ConcurrentHashMap`, an `HttpOnly` cookie. Book: Sections 10.7.1 and 10.7.2.

Program.cs: Services

Listing 10.14 From the services half of `aspnet/Program.cs`: cookie authentication, and a login required on every page but one.

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.LoginPath = "/login";                    // where anonymous visitors are sent
        options.Cookie.HttpOnly = true;
        options.Cookie.SameSite = SameSiteMode.Lax;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
    });

builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizeFolder("/");            // every page needs a login ...
    options.Conventions.AllowAnonymousToPage("/Login");  // ... except the login page
});
```

Listing 10.14: the two conventions declare once what PHP repeated in each page. Just above, `AddSingleton<IStore>` builds one store — `CsvStore`, because NuGet (and so the SQLite driver) was unreachable; `SqliteStore` compiles with `-p:UseSqlite=true`. Book: Sections 10.8.1 and 10.8.2.

Program.cs: the Pipeline

Listing 10.15 The pipeline half of `aspnet/Program.cs`: static files, routing, identity checked before permissions, then the pages.

```
// ---- the pipeline: every request passes through these, in this order ----------------------
var app = builder.Build();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");                  // one sentence, no stack trace
}
app.UseStaticFiles();                                   // wwwroot/static/style.css, the logo
app.UseRouting();
app.UseAuthentication();                                // who is this? (reads the cookie)
app.UseAuthorization();                                 // may they see this page?
app.MapGet("/", () => Results.Redirect("/customers"));
app.MapRazorPages();                                    // the @page routes of Pages/*.cshtml
app.Run();
```

There is no CSRF code: Razor Pages refuses every `POST` without a valid antiforgery token with **400**, and the form tag helper adds the token.

Listing 10.15 is a filter chain that is the whole application. Configuration is layered: JSON files, environment (`Laundry__Store=sqlite`), command line. Book: Section 10.8.2.

A Razor Page: Route, Model, View

Listing 10.16 `Customers.cshtml` (its title block left out): the same markup as the PHP and JSP views, with `@` switching to C# for one expression at a time.

```
@page "/customers"
@model CustomersModel
<h1>Customers</h1>
<form method="get" action="/customers">
  <label for="q">Name or phone contains</label>
  <input id="q" name="q" value="@Model.Q">
  <button>Search</button>
</form>
<p class="count">@n customer@(n == 1 ? "" : "s") found@(n > Model.Result.Rows.Count ? $", showing the
first {Model.Result.Rows.Count}" : "").</p>
<table>
  <tr><th>Id</th><th>Name</th><th>Phone</th><th>E-mail</th></tr>
  @foreach (var c in Model.Result.Rows)
  {
  <tr><td>@c.Id</td><td>@c.Name</td><td>@c.Phone</td><td>@c.Email</td></tr>
  }
</table>
```

Listing 10.16: `@page "/customers"` *is the route; what is not there is* `e()` *— every* `@` *expression is HTML-encoded. In* `/bills/new` *against* `/bills/{id}`*, literal segments beat parameters. Book: Section 10.8.3.*

The Login as Claims in a Cookie

Listing 10.17 The end of `OnPostAsync` in `Login.cshtml.cs`: after the same password check, four claims go into an encrypted cookie.

```
        var identity = new ClaimsIdentity(new[]
        {
            new Claim(ClaimTypes.NameIdentifier, e.Id),
            new Claim(ClaimTypes.Name, e.Name),
            new Claim(ClaimTypes.Role, Rules.Role(e)),
            new Claim("outlet", e.OutletId ?? ""),
        }, CookieAuthenticationDefaults.AuthenticationScheme);
        // A new, encrypted authentication cookie replaces whatever the browser had.
        await HttpContext.SignInAsync(new ClaimsPrincipal(identity));
        return Redirect("/customers");
```

Listing 10.17: the cookie is encrypted and signed by Data Protection; the server remembers nothing per user. `SignInAsync` replaces whatever the browser held, so session fixation has nothing to fix. Book: Section 10.8.4.

Where the Login Lives

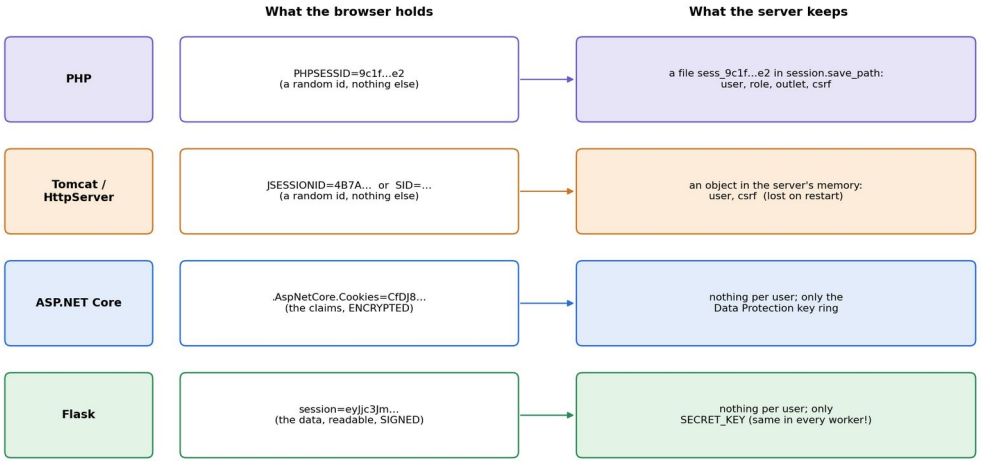


Figure 10.4 Where the login lives in each build. PHP and the Java builds give the browser only an id and keep the data on the server; ASP.NET Core and Flask put the data in the cookie, protected by a key that only the server holds.

Figure 10.4: a restart forgets no cookie-based login and many copies accept one cookie if they share the key; but the cookie is larger (560 characters against 32) and cannot be revoked by deleting a session. Book: Section 10.8.4.

A Finding: the Missing Token

Listing 10.18 The first eight lines of the acceptance test against the first version of the ASP.NET build. Check 02 found no hidden field in the login form, and every POST was refused with 400: the antiforgery token was missing.

```
PASS 01 anonymous /customers is sent to /login
     server says: ASP.NET Core Razor Pages
FAIL 02 login page has a form with a hidden token  <--
FAIL 03 wrong password: 401 and one neutral message  <-- 400
PASS 04 login without the token is refused (400 or 403)
FAIL 05 cashier logs in and lands on /customers  <-- 400 None
FAIL 06 all customers: 256 found, first 20 shown  <-- 302
FAIL 07 search 'budi' finds 3  <--
```

The first login form was written `<form method="post" action="/login">`; the tag helper treats a literal `action` as a form to another site and leaves the token out.

The fix is one attribute: `<form method="post" asp-page="/Login">` — then all eighteen checks passed.

Listing 10.18: a secure default lost silently, and caught by a test that behaves like a browser. Book: Section 10.8.5.

Flask: CSRF and the Login Decorator

Listing 10.19 From `create_app` in `python/app.py`: the token check before every POST, and the `login_required` decorator.

```
    @app.before_request
    def check_csrf():
        if request.method == "POST":
            sent = request.form.get("csrf", "")
            if not sent or not secrets.compare_digest(sent, session.get("csrf", "")):
                abort(403, "The form has expired. Go back, reload the page and try again.")

    def login_required(view):
        @functools.wraps(view)
        def wrapper(*args, **kwargs):
            if "user" not in session:
                return redirect(url_for("login"))
            return view(*args, **kwargs)
        return wrapper
```

Listing 10.19: a `context_processor` gives each session its token; `before_request` is the servlet filter; `teardown_appcontext` closes the connection like try-with-resources. `@login_required` is opt-in per view, like PHP's `require_login()`. Flask is a WSGI application (PEP 3333). Book: Sections 10.9.1 and 10.9.2.

Flask: the New-Bill View

Listing 10.20 `new_bill` in `python/app.py`: 403 for a washer, the form, 422 with the messages, or the bill and a 303.

```
@app.route("/bills/new", methods=["GET", "POST"])
@login_required
def new_bill():
    if session["user"]["role"] == "washer":
        abort(403, "Washers do not take new bills.")
    if request.method == "GET":
        return render_template("bill_new.html", services=store().services(),
                               values={"units": "1"}, errors={})
    values, errors = rules.validate_bill(request.form, store())
    if errors:
        return render_template("bill_new.html", services=store().services(),
                               values=values, errors=errors), 422
    user = session["user"]
    bill_id = store().create_bill(values["customer"], values["service"], int(values["units"]),
                                  values["rack"], user["id"], user["outlet"], now())
    return redirect(url_for("show_bill", bill_id=bill_id), code=303)
```

Listing 10.20: routes are decorators; `/bills/new` is matched as a literal before `/bills/<bill_id>`. `url_for` builds the address from the view's name, like `RedirectToPage`. Book: Section 10.9.2.

Jinja2: Escaping by Default

Listing 10.21 `templates/customers.html`: `{% extends %}` names the layout, and the page supplies only its title and its block. Every `{{ … }}` is escaped.

```
{% extends "base.html" %}
{% set title = "Customers" %}
{% block main %}
<h1>Customers</h1>
<form method="get" action="/customers">
  <label for="q">Name or phone contains</label>
  <input id="q" name="q" value="{{ q }}">
  <button>Search</button>
</form>
<p class="count">{{ total }} customer{{ "" if total == 1 else "s" }} found{% if total > rows|length %},
showing the first {{ rows|length }}{% endif %}.</p>
<table>
  <tr><th>Id</th><th>Name</th><th>Phone</th><th>E-mail</th></tr>
  {% for c in rows %}
  <tr><td>{{ c.id }}</td><td>{{ c.name }}</td><td>{{ c.phone }}</td><td>{{ c.email }}</td></tr>
  {% endfor %}
</table>
{% endblock %}
```

Listing 10.21: a page extends the layout and fills its blocks; every `{{ … }}` is escaped, and raw HTML needs the `safe` filter — the phrase to search for in a review. Book: Section 10.9.3.

Flask's Session Is Signed, Not Encrypted

```
{"user":{"id":"E0004","name":"Dewi Lestari Putri","outlet":"001","role":"cashier"}}
```

That is the first part of the cashier's session cookie, Base64-decoded: **anyone holding the cookie can read it**.

Nobody can *change* it without `SECRET_KEY`: a timestamp and an HMAC signature follow the data.

Rule: put in Flask's session only what the user may see — never anything secret.

The key is the security of every login; `create_app` invents a random one if `LAUNDRY_SECRET_KEY` is not set, which is a trap in production. Book: Section 10.9.4.

One Test From the Outside

`acceptance/acceptance.py`, standard-library Python, behaves like a **careful browser**: it keeps cookies, submits a form's hidden fields, and does not follow redirects.

It never knows the name of the anti-forgery field — `csrf` in four builds, `__RequestVerificationToken` in ASP.NET Core.

The checks are security properties: 04 and 12 CSRF, 03 a neutral login error, 09 escaping, 10 SQL injection, 17 real logout, 18 permissions by role.

None of these is visible when an application is demonstrated by clicking through it. Book: Section 10.10.1.

The Results

Table 10.4 The acceptance test in this book's build, each run against a fresh copy of the database with the clock frozen at 2026-09-30 21:00:00.

Build	Port	Result	Notes
PHP 8.4 + PDO, built-in server	9001	18 passed, 0 failed	Section 10.5.8
Java HttpServer + JDBC	9002	18 passed, 0 failed	
ASP.NET Core Razor Pages, Kestrel	9003	18 passed, 0 failed	after the fix of Section 10.8.5; the first version passed 3
ASP.NET Core, `dotnet publish` output	9003	18 passed, 0 failed	Section 10.12.3
Flask, development server	9004	18 passed, 0 failed	
Flask under gunicorn, 2 workers, with `LAUNDRY_SECRET_KEY`	9005	18 passed, 0 failed	Section 10.12.4
Flask under gunicorn, 2 workers, without it	9005	12 passed, 6 failed	Section 10.12.4
Servlet + JSP on Tomcat 11	8080	not run	Tomcat and JSTL unavailable (Section 10.6.7)

Table 10.4: every running build passed all eighteen checks — the ASP.NET build after the fix of Section 10.8.5; the servlet build could not be run here. Book: Section 10.10.2.

Where the Builds Differ on the Wire

Table 10.5 What each build actually sent, as recorded in this book's build. Every cookie was `HttpOnly` with `SameSite=Lax`; ASP.NET Core also sets an antiforgery cookie with `SameSite=Strict`.

Request	PHP	Java HttpServer	ASP.NET Core	Flask	Servlet (from the code)
`GET /customers`, not logged in	303 → `/login`	303 → `/login`	302 → `/login?ReturnUrl=%2Fcustomers`	302 → `/login`	302 → `/login`
`POST /login`, right password	303	303	302	303	303
`POST /bills/new`, valid	303	303	302	303	303
any `POST` without the token	403	403	400	403	403
login cookie	`PHPSESSID`, 32 characters	`SID`, 64 characters	`.AspNetCore.Cookies`, 560 characters, encrypted	`session`, signed data	`JSESSIONID`
one login, measured once	0.16 s	0.24 s	0.18 s	0.24 s	—

Table 10.5: after a POST, 303 See Other says what is meant; 302 works in every browser. A missing token gets 403 from hand-written checks and 400 from ASP.NET Core. A `ReturnUrl` must be checked, or it is an open redirect. Book: Section 10.10.3.

From an Address to Code

Table 10.6 How each build turns an address into code. Three of the five put the route next to the code it runs; two keep all routes in one place.

Build	Where the routes are written	Form of one route	When two could match
PHP	one table in `public/index.php`	`['GET', '#^/bills/([A-Za-z][0-9]{4})$#', 'show_bill']`	the first row in the table wins
Servlet	an annotation on each class	`@WebServlet("/bills/*")`	exact beats path prefix beats extension beats default
HttpServer	a `switch` in `dispatch`	`case "GET /customers" -> …`	the `switch`, then the regex in `default`
ASP.NET Core	a directive at the top of each page	`@page "/bills/{id}"`	literal segments beat parameters
Flask	a decorator on each function	`@app.get("/bills/<bill_id>")`	rules sorted by specificity; static parts first

Table 10.6: all routes in one place (PHP, HttpServer) or each route beside its code (servlets, Razor, Flask). Every mechanism lets the fixed `/bills/new` win over the pattern; PHP relies on your ordering. Book: Section 10.11.1.

Templates and Escaping

Table 10.7 Templates in the five builds. The last column is the one to search for when reviewing code for cross-site scripting.

Build	Template language	Layout mechanism	Escaping	How to print raw HTML
PHP	PHP itself (`<?= ?>`)	capture the page, then include `layout.php`	**opt-in**: `e()`	just `<?= $x ?>`
Servlet + JSP	JSP with EL and JSTL	static include of `_top.jspf` and `_bottom.jspf`	**opt-in**: `<c:out>`	just `${x}`
HttpServer	Java text blocks	a `layout` method	**opt-in**: `esc()`	just `%s`
ASP.NET Core	Razor (`@`)	`_Layout.cshtml` with `@RenderBody()`	**automatic**	`@Html.Raw(x)`
Flask	Jinja2 (`{{ }}`, `{% %}`)	`{% extends %}` with blocks	**automatic**	the `safe` filter

Table 10.7 holds the chapter's most important security difference: in three builds a forgotten escape is an XSS hole that looks like correct code; in two, forgetting is safe. Choose a template engine that escapes by default. Book: Section 10.11.2.

The Four Systems Compared

Table 10.8 The four systems of the chapter compared: language, process model, entry point, checks, login state and CSRF protection.

	PHP 8 + PDO	Servlet + JSP	ASP.NET Core Razor Pages	Flask
Language	PHP 8.4	Java 17+	C# 14 on .NET 10	Python 3.10+
Process model	a fresh start per request (shared-nothing)	one long-lived JVM, many threads	one long-lived process, many threads	one or more long-lived processes, one request each at a time (sync workers)
Entry point	`public/index.php`	the container reads the WAR	`Program.cs`	`create_app()`
Cross-cutting checks	top of the front controller	a `Filter`	middleware and conventions	`before_request`, decorators
Login state	file per session on the server	object per session in memory	encrypted claims in the cookie	signed data in the cookie
CSRF protection	hand-written	hand-written	built in, automatic	hand-written (or Flask-WTF)

Table 10.8: the process model explains most of the other rows. Database APIs, configuration and deployment are compared in the book's full table. Book: Section 10.11.3.

A Front Server Before Every App

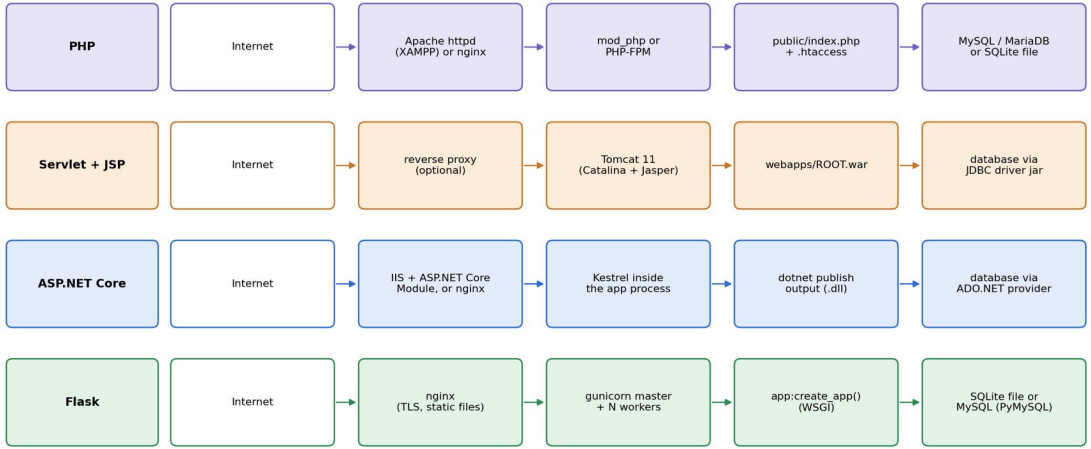


Figure 10.5 A production arrangement for each system. In every row a front web server handles TLS, static files and slow clients, and hands the rest to the program that runs the application.

Figure 10.5: Apache, nginx or IIS holds the TLS certificate, serves static files and copes with slow clients; the application server runs your code. PHP deploys by copying files, Java as a WAR, ASP.NET Core as the `dotnet publish` folder. Book: Sections 10.12–10.12.3.

gunicorn, and a Real Deployment Bug

gunicorn's starting point for the number of workers:

$$\text{workers} = 2 \times \text{CPU cores} + 1 \tag{10.3}$$

Without `LAUNDRY_SECRET_KEY`, each worker ran `create_app` and **invented its own key**: a cookie signed by one worker was a forgery to the other. The test failed six checks — different ones on each run.

> **THE LESSON.** Anything a process invents at start-up is different in every process. Set one secret key for every worker; ASP.NET Core instances must share the Data Protection key ring for the same reason.

By Equation (10.3) the two-core build machine would take five workers; two were enough. Under the one-process development server the bug cannot appear, which is exactly why it reaches production. Book: Section 10.12.4.

Which to Choose

Table 10.9 A starting point for choosing. Every row assumes a team that knows the language; if that assumption fails, it outweighs everything else in the table.

If your situation is mostly…	Consider	Because
cheap shared hosting, a PHP-literate team, content-heavy sites	PHP (with Laravel or Symfony when it grows)	deployment by copying files; the largest hosting market
existing Java systems, large teams, long-lived enterprise software	Jakarta EE / Spring on Tomcat	strong typing, mature tooling, the enterprise ecosystem
a Microsoft environment, SQL Server, Windows servers, or strong defaults wanted	ASP.NET Core	secure defaults, excellent tooling, one runtime on every OS
data analysis, AI or scripting already in Python; a small, fast-moving team	Flask, or Django for more structure	one language for the web and the data

Table 10.9: for a shop's back office all four are adequate. What decides is what the people know, what the application must connect to, where it will run, how much the framework does for you, and a supported version. Book: Section 10.13.

Profitina in Practice

The choice followed the connections: scanning, scoring and a local Qwen2.5 model live in Python, so the web application is FastAPI on Uvicorn (ASGI), with PostgreSQL.

Front and application server apart: Cloudflare Tunnel faces the Internet; the app runs as a Windows service under NSSM.

An acceptance gate: a release counts only when `/api/health` answers and over a thousand tests are green; passwords are Argon2id hashes.

The runnable case study remains Profitina Laundry; what carries across is one contract, several implementations, and a test that holds each to it. Book: Section 10.14.

Common Mistakes

Escaping and forms: forgetting `e()` in one PHP, JSP or text-block view; a literal `action` on a Razor Pages form.

Deployment: a random secret key per process; a `jakarta` WAR on a `javax` Tomcat; trusting the development server.

Code: request data in a servlet field; route order without knowing it; `match` instead of `fullmatch`; SQL built from the search text.

Behavior: no redirect after a POST; an unchecked `ReturnUrl`; saying which half of a login was wrong; a demonstration instead of a test.

Each mistake is listed in the book with its cure. Book: Section 10.15.

Summary: Five Things to Remember

1. One contract, five builds, the same six stages: accept, check, route, decide, data, view.

2. Keep the rules and the password check outside the framework; iterations make guessing expensive.

3. PHP starts from nothing; servlets live long; ASP.NET Core and Flask give escaping by default.

4. Sessions come in two families — an id with data on the server, or data in a protected cookie that needs one shared key.

5. A test from the outside proves the refusals too; choosing is about people, connections and hosting.

Book: Section 10.16.

Next Time

Chapter 11 — Dynamic Pages and Web Forms. What makes a page dynamic, single-file and code-behind pages, and the web controls in their modern form.

Before then: work Exercises 10.2, 10.3, 10.5 and 10.6.

The solutions to all twelve exercises are in the book. Book: Sections 10.17 and 10.18.

Lecture 11 — Dynamic Pages and Web Forms

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 11

Lecturer: Irfan Subakti — yifana@gmail.com

Today we look closely at the thing every earlier chapter used without examining: the form — what it sends, byte by byte, how Web Forms hid it, and how a real form is built safely. Book: Chapter 11, Sections 11.1–11.16.

What This Meeting Covers

1. **Static and dynamic pages** — when content is decided; a visitor counter and its lost update.
2. **Where the code lives** — inline, single-file and code-behind; a page is a class.
3. **HTML forms in depth** — attributes, successful controls, GET and POST, input types, three encodings, uploads.
4. **The Web Forms model** — server controls, postback, ViewState, the life cycle, events.
5. **Razor Pages forms** — model binding, tag helpers, handlers, validation, Post/Redirect/Get, antiforgery.
6. **Profitina Laundry** — the new-bill form and a receipt upload, in four technologies, and the security of forms.

Web Forms runs only on Windows with the .NET Framework 4.8; its listings are ideas, marked "not run", each followed by a Razor Pages form that was run. Book: Section 11.1.

Where We Left Off: the Form

Chapter 9 introduced ASP.NET and Razor Pages; Chapter 10 ran PHP, JSP and ASP.NET side by side; Chapter 7 built a Laundry form in servlets and JSP.

Logging in, searching, taking in laundry, paying, uploading a photo: **forms, every one**.

A programmer who does not know exactly what a form sends writes applications that are wrong in ways nobody notices — until a customer, or an attacker, does.

Four questions: what makes a page dynamic; what a form sends; how Web Forms hid HTTP and what replaced it; how a real form is built safely and tested. Book: Section 11.1.

Static Pages and Dynamic Pages

> **DEFINITION.** A **static page** is a file whose bytes are decided before anyone asks for it; the server sends it unchanged to every visitor. A **dynamic page** is produced by a program at the moment it is requested, so two requests for the same address can receive different bytes.

A static `VisitorCounter01.html` says *visitor number 1* to every visitor, forever: counting needs a memory that changes with every request.

Neither kind is better: the price list could be static; anything that depends on who asks, what they typed or the database is dynamic.

Any plain HTML file is static; an `.aspx`, `.php`, `.jsp` or `.cshtml` page is dynamic. Book: Sections 11.2.1 and 11.2.2.

Two Kinds of Page

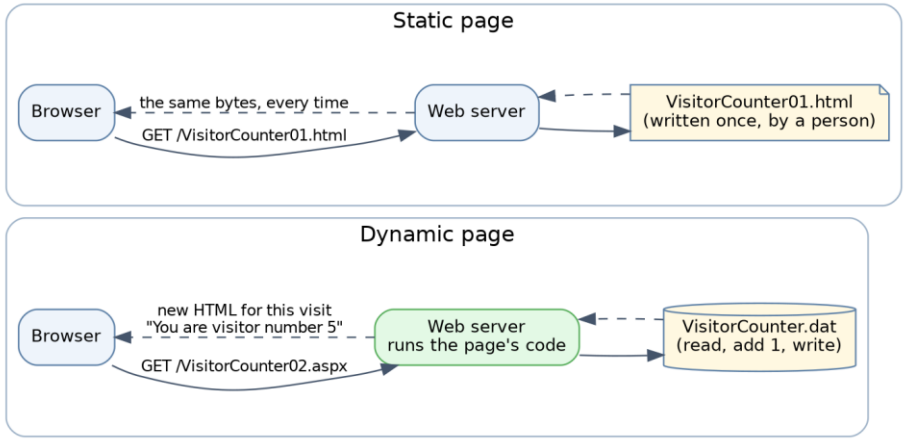


Figure 11.1 A static page and a dynamic page. The static page is copied from disk and is the same for everyone; the dynamic page runs the page's code, which reads and updates a counter file and writes new HTML for each visit.

*In **Figure 11.1** the server's only work for a static page is to find a file and copy it out; for a dynamic page it runs code that reads and updates a counter file and writes HTML that did not exist a moment earlier. Book: Section 11.2.1.*

The Counter, Safely, in Razor Pages

Listing 11.1 The page model of `Counter.cshtml.cs`: `OnGet` runs for every GET; the `lock` makes read, add and write one indivisible step.

```
public class CounterModel(IWebHostEnvironment env) : PageModel
{
    static readonly object Gate = new();          // one request at a time reads and writes the file
    public int Visitor { get; private set; }

    public void OnGet()
    {
        string file = Path.Combine(env.ContentRootPath, "var", "VisitorCounter.dat");
        lock (Gate)
        {
            Directory.CreateDirectory(Path.GetDirectoryName(file)!);
            int n = System.IO.File.Exists(file) && int.TryParse(System.IO.File.ReadAllText(file), out int old) ?
old : 0;
            Visitor = n + 1;
            System.IO.File.WriteAllText(file, Visitor.ToString());
        }
    }
}
```

> **Read, change, write is never safe by default on a web server.** Two requests can interleave between the read and the write: a *lost update*. Protect it with a lock, a transaction or an atomic operation.

***Listing 11.1**: the Web Forms original read, added and wrote a file with no protection; two visitors in the same millisecond both read 41 and both wrote 42. Book: Sections 11.2.3 and 11.2.4.*

Fifty Visitors at Once

Listing 11.2 The counter after 4 visits and 50 simultaneous requests: the 55th visitor is visitor 55. No visit was lost.

```
You are visitor number 55
You are visitor number 56
You are visitor number 57
You are visitor number 58
```

After 4 visits, 50 requests fired ten at a time with `xargs -P 10 curl`, and one more: $4 + 50 + 1 = 55$. No visit was lost.

A `lock` protects one server process; behind a load balancer the counter must live in a database and be incremented there, in one statement.

Listing 11.2 *is exactly how the Laundry numbers its bills: in the database, not in memory. Book: Section 11.2.4.*

Code-Behind: Two Files, One Class

Listing 11.3 The counting method in `MyVisitor2.aspx.cs`, the code-behind of a class derived from `System.Web.UI.Page`. Web Forms; not run here.

```
    public int MyCounter() {
        FileStream s =
            new FileStream(Server.MapPath("~/App_Data/") +
                "VisitorCounter.dat",
                FileMode.OpenOrCreate);
        int n;
        try {
            BinaryReader r = new BinaryReader(s);
            n = r.ReadInt32();
        } catch {
            n = 0;
        }
        n++;
        s.Seek(0, SeekOrigin.Begin);
        BinaryWriter w = new BinaryWriter(s);
        w.Write(n);
        s.Close();
        return n;
    }
```

***Listing 11.3**: the page `MyVisitor2.aspx` says `Inherits="MyVisitor2"` and calls `<%=MyCounter()%>`; the C# lives in its own file, where the compiler, the debugger and tests can reach it. Book: Section 11.3.2.*

A Page Is a Class in Disguise

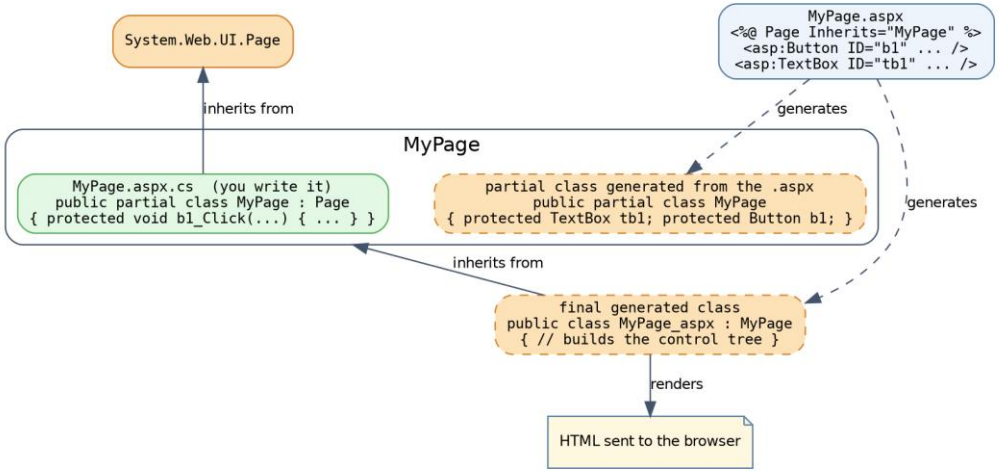


Figure 11.2 The code-behind page class. `MyPage` is one class written in two parts: your `.aspx.cs` file and a part generated from the markup, which declares a field per control. A final generated class derives from `MyPage` and renders the page.

***Figure 11.2**: `MyPage` is one `partial` class in two parts — yours, and one generated with a field per control; a final generated class renders the page. Every request is served by a new object, which knows nothing about the previous request. Razor Pages keeps the split: `.cshtml` plus a `PageModel`. Book: Sections 11.3.3 and 11.3.4.*

The Smallest Useful Form

Listing 11.4 `MyAccount.html`: a form with two text fields and a submit button, posted to `/echo`.

```
<html>
    <head>
        <title>My Account</title>
    </head>
    <body>
        <form action="/echo" method="post">
            <b>Balance:</b>
            <input type="text" name="total" value="0">Euro<br>
            <input type="text" name="amount">
            <input type="submit" name="OK" value="Pay">
        </form>
    </body>
</html>
```

***Listing 11.4**: the `<form>` element carries the instructions for sending, the controls carry the data. Filled with 10 and 25, `/echo` received `total=10`, `amount=25` and `OK=Pay`: the button sent its own name because it was pressed; Balance: and Euro are not data. Book: Section 11.4.1.*

The Attributes of form

Table 11.1 The attributes of `<form>`. A submit button may override four of them for itself with `formaction`, `formmethod`, `formenctype` and `formnovalidate`.

Attribute	Values	What it decides
`action`	a URL; empty or missing means the page's own address	where the request goes
`method`	`get` (the default), `post`, `dialog`	the HTTP method, and so whether the data goes into the URL or the body
`enctype`	`application/x-www-form-urlencoded` (default), `multipart/form-data`, `text/plain`	how a `POST` body is encoded (Section 11.4.5)
`accept-charset`	normally `UTF-8`	the character encoding of the values; the page's own encoding if missing
`novalidate`	present or absent	switches off the browser's own checks (Section 11.4.7)
`target`	`_self`, `_blank`, a frame name	which window shows the answer
`autocomplete`	`on`, `off`	whether the browser offers remembered values
`name`	text	a name for scripts; never sent

***Table 11.1**: a button's own `formaction` wins over the form's `action` — Razor Pages uses exactly this for its named handlers. Book: Section 11.4.1.*

What a Form Sends: Successful Controls

Table 11.2 Which controls a browser sends, first part. The row that causes most bugs is the checkbox: unchecked, it sends *nothing*.

Control	Sent when	Sent as
any control without a `name`	never	—
any `disabled` control	never	—
`readonly` text	always	`name=value`
text, email, number, date, … , `textarea`	always, even when empty	`name=` for an empty field
checkbox	only when checked	`name=value`, or `name=on` without a `value`
radio button	only the checked one of its group	`name=value`
`select`	always, the selected options	one pair per selected option

***Table 11.2**: an unchecked checkbox sends nothing, so a missing name must mean off; a repeated name arrives several times. Book: Section 11.4.2.*

Successful Controls, Continued

Table 11.3 Which controls a browser sends, second part. Only the submit button that was pressed is sent.

Control	Sent when	Sent as
`hidden`	always	`name=value`
submit button	only the one that was pressed	`name=value`
`input type=image`	only when clicked	`name.x=12&name.y=7`, the click position
`reset`, `type=button`	never	—
file	always in `multipart/form-data`	the file name *and its bytes*; only the name otherwise

***Table 11.3**: only the pressed button is sent, which is how one form with two buttons tells the server which was meant; a file travels with its bytes only in `multipart/form-data`. Book: Section 11.4.2.*

GET and POST on the Wire

Listing 11.5 A GET request: the data is in the request line, visible in the address bar, in the history, and in every server log.

```
0000: GET /echo?customer=C0047&note=Kebaya+%26+batik HTTP/1.1
0039: Host: 127.0.0.1:9104
004f: User-Agent: curl/8.5.0
0067: Accept: */*
0074:
```

Listing 11.6 The same data as a POST request: the pairs move into the body, and two headers describe the body — its length and its encoding.

```
0000: POST /echo HTTP/1.1
0015: Host: 127.0.0.1:9104
002b: User-Agent: curl/8.5.0
0043: Accept: */*
0050: Content-Length: 36
0064: Content-Type: application/x-www-form-urlencoded
0095:
0000: customer=C0047&note=Kebaya+%26+batik
```

Listing 11.5 *and* ***Listing 11.6*** *are the real bytes from* `curl --trace-ascii`*. The pairs are identical,* `customer=C0047&note=Kebaya+%26+batik`*; only their address changed — request line or body. Book: Section 11.4.3.*

GET and POST Compared

Table 11.4 GET and POST compared. The rule of thumb: if pressing the button twice would be a problem, it must be POST.

	GET	POST
where the data goes	the URL's query string	the request body
meaning in HTTP	*safe*: "give me something"; must change nothing	"process this"; may change things
repeating it	harmless (it is also *idempotent*)	may create a second bill
bookmark, share, back button	yes, the whole request is the URL	no; Reload asks "resubmit the form?"
cached by browsers and proxies	may be	not by default
visible in logs and history	yes	no (the body is not logged by default)
size	limited by URL length, a few thousand characters in practice	limited only by the server's setting
can carry a file	only its name	yes, with `multipart/form-data`
use it for	searching, filtering, paging	creating, changing, deleting, logging in, uploading

***Table 11.4**: never send a password, or anything that changes data, with GET. The URL ends up in history, bookmarks, the `Referer` header and every log; sent as GET, the test form put `pin=4321` in the address bar. Book: Section 11.4.3.*

Every Input Type on One Page

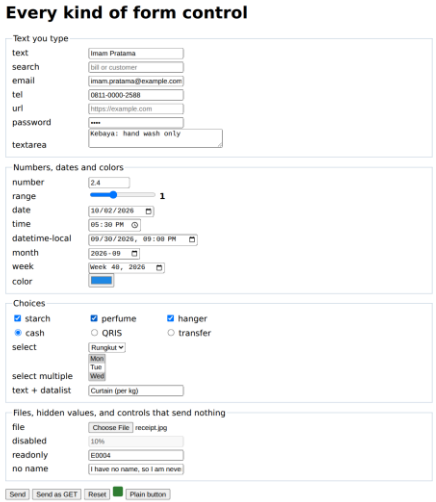


Figure 11.3 The test page in headless Chromium: text types, numbers and dates, choices, a file, and three controls that send nothing — *disabled*, the control without a name, and the *Reset* and *Plain button* buttons.

***Figure 11.3**: the browser draws calendars, clocks, color pickers and sliders by itself — but **what arrives is always text** in a fixed format: `2026-10-02` for a date, `2.4` with a dot for a number, whatever the display shows. Book: Section 11.4.4.*

Encoding the Body: urlencoded

Listing 11.7 The urlencoded body as Chromium sent it, then as `Forms.urlencoded` parsed it. The ampersand became `%26`, the comma `%2C`, each space `+`, and the one character ½ became two bytes, `%C2%BD`.

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 62

customer=C0047&note=Kebaya+%26+batik%2C+2%C2%BD+kg&express=yes

--- parsed ---
customer = [C0047]
note = [Kebaya & batik, 2½ kg]
express = [yes]
```

Every byte that is not a letter, a digit or `-._~` becomes `%XX`; a space becomes `+`. With n_s bytes kept and n_e escaped:

$$L = n_s + 3\,n_e \tag{11.1}$$

> **WORKED EXAMPLE.** `Kebaya & batik, 2½ kg` has 22 bytes (½ is two). The ampersand, the comma and the two bytes of ½ are escaped: $n_e = 4, n_s = 18, L = 18 + 3 \times 4 = 30$.

***Listing 11.7**; by Equation (11.1) a photo, nearly all "unsafe" bytes, would almost triple — which is why nobody sends files this way. Book: Section 11.4.5.*

Encoding the Body: multipart

Listing 11.8 The multipart body as Chromium sent it: one part per field, separated by the boundary; the file's bytes travel exactly as they are.

```
Content-Type: multipart/form-data; boundary=----WebKitFormBoundarypOUphwA3PZopjEM9
Content-Length: 404

------WebKitFormBoundarypOUphwA3PZopjEM9\r
Content-Disposition: form-data; name="customer"\r
\r
C0047\r
------WebKitFormBoundarypOUphwA3PZopjEM9\r
Content-Disposition: form-data; name="note"\r
\r
Kebaya & batik, 2.. kg\r
------WebKitFormBoundarypOUphwA3PZopjEM9\r
Content-Disposition: form-data; name="photo"; filename="tag.txt"\r
Content-Type: text/plain\r
\r
rack A03
\r
------WebKitFormBoundarypOUphwA3PZopjEM9--\r
```

***Listing 11.8**: instead of escaping, multipart chooses a boundary that occurs in no value and announces it in `Content-Type`; a 2 MB photo costs 2 MB plus a few hundred bytes. RFC 7578 defines it. Book: Section 11.4.5.*

The Three Encodings

Table 11.5 The three form encodings. A form with a file input and the default enctype sends only the file's name, silently.

enctype	Separator	Escaping	Files	Use
`application/x-www-form-urlencoded`	`&` and `=`	`%XX`, space as `+`	name only	every form without a file (the default)
`multipart/form-data`	a boundary line	none; raw bytes	yes	any form with `<input type="file">`
`text/plain`	line breaks	none	no	nothing a program should parse

***Table 11.5**: `text/plain` writes `name=value` lines with no escaping — for people reading e-mail, not for programs. An upload needs `method="post"` and `enctype="multipart/form-data"`. Book: Sections 11.4.5 and 11.4.6.*

Uploads, and Why Browser Checks Are Not Enough

An uploaded part carries a **file name**, a **content type** and the **bytes**. Only the bytes are facts; the name and type are chosen by the sender.

Every server caps the request size: PHP's `upload_max_filesize`, Flask's `MAX_CONTENT_LENGTH`, ASP.NET Core's `[RequestSizeLimit]`, the Java track's `MAX_BODY`.

`required`, `pattern`, `min`, `max` help an honest user — and are removed in seconds with the developer tools, `novalidate` or `curl`.

> **THE KEY INSIGHT.** Browser validation is a convenience for honest users; server validation is the only validation. The server must also check what the browser cannot know — that the customer exists, that the rack is free, what the price is.

`capture="environment"` *asks a phone to open the rear camera — what a cashier photographing a signed receipt wants. Book: Sections 11.4.6 and 11.4.7.*

The Problem Web Forms Set Out to Solve

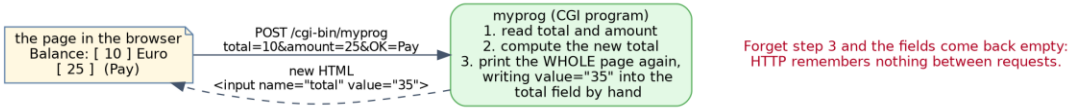


Figure 11.4 One CGI round trip. The program receives the fields, computes, and must write the entire page again — including every value the user had typed — or the user sees an empty form.

***Figure 11.4**: HTTP remembers nothing, so a CGI program must write the whole page again, copying back every typed value, correctly escaped. Desktop programmers had objects with a `Click` event; Web Forms set out to give web programmers the same. Book: Section 11.5.*

Server Controls in a Tree

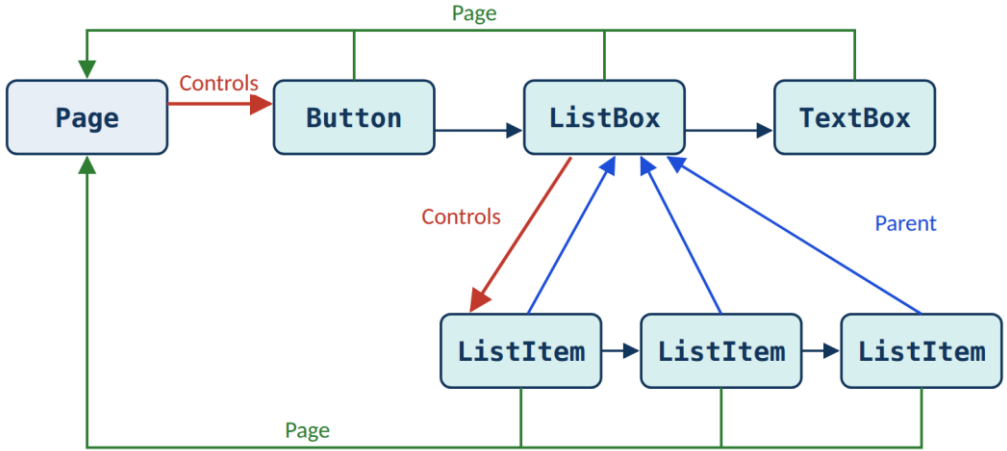


Figure 11.5 The control tree of a page with a `Button`, a `ListBox` with three `ListItem` children, and a `TextBox`. Red arrows are `Controls` (down the tree), blue arrows `Parent` (up one level), green arrows `Page` (straight to the root), navy arrows the order inside a `Controls` collection.

*`<asp:Label ID="total" Text="Hello" runat="server" />` becomes an object of class `Label` for each request; code writes `total.Text = "35";` and the control renders itself as `<span>`. **Figure 11.5**: `Controls` down the tree, `Parent` up, `Page` to the root. Book: Sections 11.6.1 and 11.6.2.*

Event Handlers That Never Touch HTTP

Listing 11.9 From `MyAccount.aspx.cs`: two handlers read and write controls as objects. Web Forms; not run here.

```
public partial class MyAccount : System.Web.UI.Page {
    protected void calculateAccount(object sender, EventArgs e) {
        int total = 1000;
        int value = Convert.ToInt32(TextBox1.Text);
        Label1.Text = (total + value).ToString();
    }
    public void DoCommand(object sender, CommandEventArgs e) {
        double total = Convert.ToDouble(myLabel.Text);
        if (e.CommandName == "add") {
            total += total * Convert.ToDouble(e.CommandArgument);
        } else if (e.CommandName == "sub") {
            total -= total * Convert.ToDouble(e.CommandArgument);
        }
        myLabel.Text = total.ToString("F2");
    }
```

***Listing 11.9**: type 10, press Calculate, and the label shows 1010; + 10% takes 100 to 110.00. Nothing parses a request, and nothing copies a typed value back — the promise Web Forms made. Book: Section 11.6.3.*

Postback

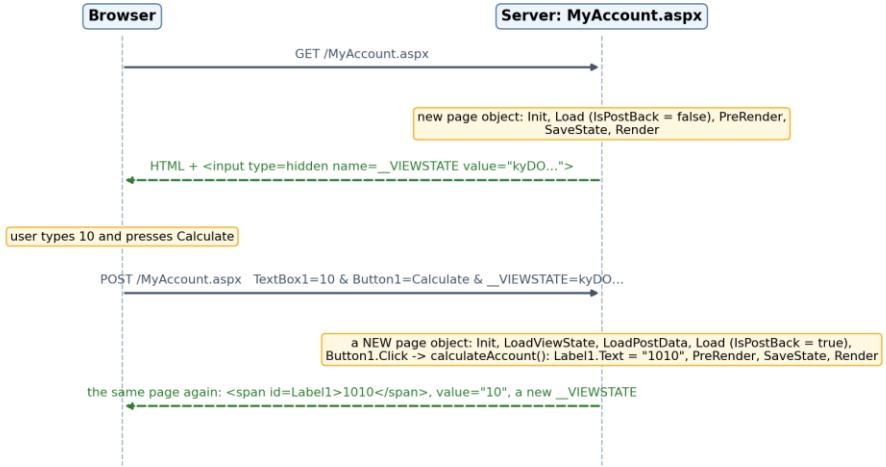


Figure 11.6 A postback. The first GET builds a page object and renders it with ViewState; pressing *Calculate* posts the whole form back; a *new* page object restores the ViewState, applies the posted values, raises `Click`, and renders the page again.

> **DEFINITION.** A **postback** is a `POST` in which a page's form is submitted to the same page. `Page.IsPostBack` is `false` on the first request and `true` on every postback.

***Figure 11.6**: the rendered page carries hidden `__VIEWSTATE`, `__EVENTTARGET` and `__EVENTVALIDATION`; `__doPostBack` lets a radio button submit the form. Book: Section 11.6.4.*

ViewState and What It Costs

Before rendering, every control's changed state is serialized, protected by a MAC, Base64-encoded into `__VIEWSTATE`, and restored on the postback. For n bytes of state:

$$L_{64} = 4\lceil \frac{n}{3} \rceil \tag{11.2}$$

WORKED EXAMPLE. A `GridView` of 300 bills of 100 bytes: $n = 30000$, so Equation (11.2) gives $L_{64} = 40000$ characters — in the page, and again in every click, even a click that only sorts the list.

ViewState is readable. Base64 is an encoding, not encryption; never switch off its MAC, and keep secrets out of it.

The label "remembers" 1010 because its memory made the round trip inside the page. Book: Section 11.6.5.

The Web Forms Page Life Cycle

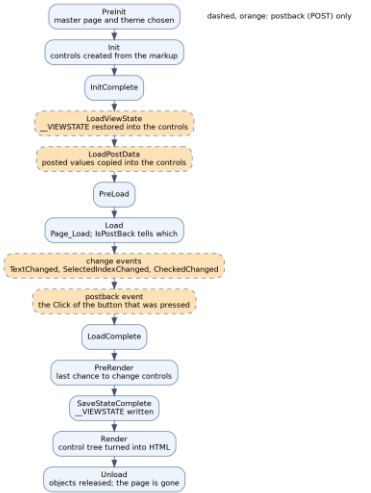


Figure 11.7 The Web Forms page life cycle. On a postback the page first restores ViewState and the posted values, then raises the change events and finally the event of the control that caused the postback.

Figure 11.7*: posted values are loaded before* `Load`*;* `Click` *runs after* `Load`*. A* `Page_Load` *that refills a list on every request, without* `if (!IsPostBack)`*, wipes the user's selection before* `Click` *can read it. Change events are queued until the next postback, unless* `AutoPostBack="true"`*. Book: Sections 11.6.6 and 11.6.7.*

Web Forms Ideas, Razor Pages Forms

Table 11.6 Web Forms ideas and their Razor Pages equivalents, first part: files, controls, properties, forms, buttons and the first visit.

Web Forms	Razor Pages	What changed
`Page.aspx` + `Page.aspx.cs`	`Page.cshtml` + `Page.cshtml.cs`	the same split; the model is a `PageModel`
server control `<asp:TextBox>`	plain `<input>` with a **tag helper**, `asp-for`	you write the HTML; the helper fills in `name`, `id`, `value`
control property `TextBox1.Text`	page-model property with `[BindProperty]`	**model binding** copies posted values into properties
one server form, always posting to itself	any number of ordinary forms	`action` and `formaction` are yours again
`OnClick="calculateAccount"`	handler method `OnPostCalculate`	a button names its handler with `asp-page-handler`
`CommandName` / `CommandArgument`	handler parameters bound from the button's route values	`asp-route-name`, `asp-route-argument`
`Page_Load`, `IsPostBack`	`OnGet` for the first visit, `OnPost…` for posts	the HTTP method *is* the difference

***Table 11.6**: Web Forms did not make the web stateful; it carried state in a hidden field and rebuilt the page. Razor Pages kept the ideas that worked and dropped those that fought the web. Book: Section 11.7.1.*

Web Forms Ideas, Continued

Table 11.7 Web Forms ideas and their Razor Pages equivalents, second part. The biggest change is the first row: nothing is kept in a hidden field unless you put it there.

Web Forms	Razor Pages	What changed
ViewState	nothing hidden: what the form posts, plus hidden fields you choose	state is explicit and small
`AutoPostBack="true"`, `__doPostBack`	one line of JavaScript, `requestSubmit()`	visible in the markup
validator controls	data annotations and `ModelState`	Chapter 14
`__EVENTVALIDATION`, `ViewStateMac`	the **antiforgery token**	protects the request, not the state
`Response.Redirect`	`RedirectToPage`, the Post/Redirect/Get pattern	Section 11.8.6
page life cycle events	page filters: `OnPageHandlerSelected`, `…Executing`, `…Executed`	Section 11.7.3

***Table 11.7**: state is explicit and small; the antiforgery token protects the request, not the state. Book: Section 11.7.1.*

MyAccount Rebuilt: Handlers

Listing 11.10 Three handlers of `Account.cshtml.cs`: each Web Forms event became a method named `OnPost` plus the handler name.

```
    public void OnPostCalculate()                                   // OnClick="calculateAccount"
    {
        int total = 1000;
        Label1 = int.TryParse(TextBox1, out int value) ? (total + value).ToString() : "Not a whole number";
    }

    public void OnPostCommand(string name, decimal argument)        // OnCommand="DoCommand"
    {
        if (name == "add") Amount += Amount * argument;
        else if (name == "sub") Amount -= Amount * argument;
        Amount = Math.Round(Amount, 2);
        ModelState.Remove(nameof(Amount));   // or the hidden input would show the POSTED value again
    }

    public void OnPostBuy()                                         // OnClick="DoClick"
    {
        Bought = "You bought: " + (Apples ? "Apples " : "") + (Pears ? "Pears " : "") + (Bananas ? "Bananas " :
"");
    }
```

***Listing 11.10**: a changed bound value needs `ModelState.Remove`, or the posted value is shown again. Book: Section 11.7.2.*

Keeps Nothing Unless Told To

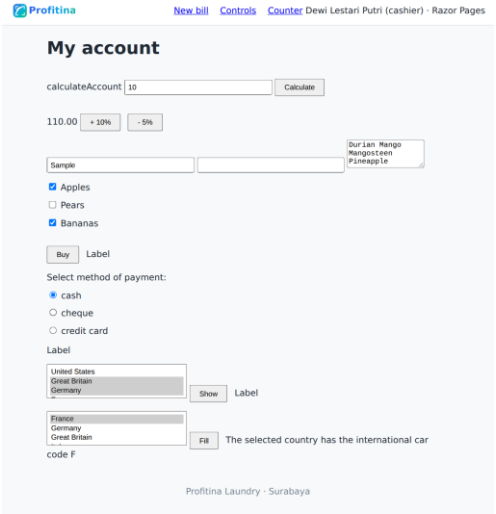


Figure 11.8 The Razor Pages `MyAccount` after seven posts. Everything the browser posts survived — the typed 10, the ticks, the radio button, both selections, the filled list — and so did the amount 110.00 carried in a hidden field. The labels of earlier posts are back to `Label`, because nothing carried them.

***Figure 11.8**: everything the browser posts survived; 110.00 survived in a hidden field on purpose; the labels of earlier posts are back to `Label`. Web Forms keeps everything unless told not to; Razor Pages keeps nothing unless told to. Book: Section 11.7.2.*

A Form Post Through Razor Pages

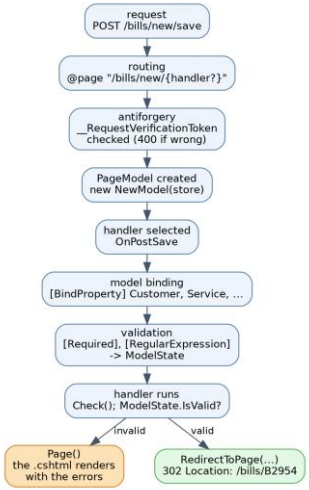


Figure 11.9 A form post through Razor Pages. Routing picks the page, antiforgery checks the token, the handler is chosen from the address, model binding and validation fill the page model, and the handler either redraws the page with errors or redirects.

***Figure 11.9**: routing (`@page "/bills/new/{handler?}"`), antiforgery (400 without a token), a new page model with its services, handler selection (`save` → `OnPostSave`), model binding and validation, then the handler. Book: Section 11.8.1.*

Model Binding: the Bound Properties

Listing 11.11 The first bound properties of `NewModel`: each is filled from the posted field of the same name; the attributes after `BindProperty` are validation rules.

```
public class NewModel(ILaundryStore store) : PageModel
{
    // Model binding fills these from the posted form, by name, before a POST handler runs.
    [BindProperty, Required(ErrorMessage = "Enter a customer id such as C0047."),
     RegularExpression(@"^\s*[Cc]\d{4}\s*$", ErrorMessage = "Enter a customer id such as C0047.")]
    public string? Customer { get; set; }

    [BindProperty, Required(ErrorMessage = "Choose a service from the list.")]
    public string? Service { get; set; }

    [BindProperty]
    public bool Express { get; set; }

    [BindProperty, Required(ErrorMessage = "Enter the weight or count, for example 2.5.")]
    public string? Quantity { get; set; }
```

***Listing 11.11**: binding looks in the form, the route values, the query string and the files, by name, ignoring case, and converts the text to the property's type. Book: Section 11.8.2.*

Four Details of Binding

Everything is text first. `Quantity` is a `string` on purpose: as a `decimal`, `2,4` would become 2.4 on an Indonesian server and 24 on an American one.

Conversion failures are not exceptions. The error goes into `ModelState`; the handler must look at `ModelState.IsValid`.

Bind only what the form may change. `[BindProperty] public Bill Bill` would let anyone add `Bill.Payment=1`: *overposting*.

GET is not bound by default. A search page says `[BindProperty(SupportsGet = true)]`.

The test suite posts `payment=1` and `price=1` with a real bill; they are ignored, and the bill costs what the price list says. The checkbox tag helper adds a hidden `Express=false`. Book: Sections 11.8.2 and 11.8.3.

Two Buttons, Two Handlers

Listing 11.12 `OnPostSave`: check first, then either redraw with errors (400) or store, put the flash in `TempData`, and redirect.

```
    /// <summary>The "Save bill" button: check, store, then redirect (Post/Redirect/Get).</summary>
    public IActionResult OnPostSave()
    {
        int units = Check();
        if (!ModelState.IsValid) { Response.StatusCode = 400; return Page(); }
        var (serviceId, total, text) = BillRules.Price(store, Service!, units, Express);
        string id = store.Save(Customer!, serviceId, units, total, Rack!, Outlet!, BillRules.Cashier,
                               BillRules.Now());
        TempData["Flash"] = $"Bill {id} saved: {text}.";
        return RedirectToPage("/Bills/Details", new { id });
    }
```

Listing 11.12: `<button asp-page-handler="Save">` *became* `formaction="/bills/new/Save"`. `OnPostPreview` *does the same checks and only quotes; it comes first in the form, so Enter never stores. Book: Sections 11.8.4 and 11.9.2.*

Validation That Needs the Shop's Data

Listing 11.13 The quantity in `Check`: tenths, the range, then kilograms rounded up or a whole number of items.

```
        int units = 0;
        if (Quantity is not null)
        {
            int? tenths = BillRules.Tenths(Quantity);
            if (tenths is null)
                ModelState.AddModelError(nameof(Quantity), "Enter the weight or count, for example 2.5.");
            else if (tenths < 1 || tenths > 1000)
                ModelState.AddModelError(nameof(Quantity), "Enter a value from 0.1 to 100.");
            else if (offered is not null)
            {
                string word = BillRules.UnitWord(offered.Description);
                if (word == "kg") units = (tenths.Value + 9) / 10;          // round UP to a whole kilogram
                else if (tenths % 10 != 0)
                    ModelState.AddModelError(nameof(Quantity), $"A whole number of {word}s, for example 2.");
                else units = tenths.Value / 10;
            }
        }
```

***Listing 11.13**: attributes check shapes; `ModelState.AddModelError` adds what needs data — an unknown customer, express on the wrong service, an occupied rack. Book: Section 11.8.5.*

Post/Redirect/Get

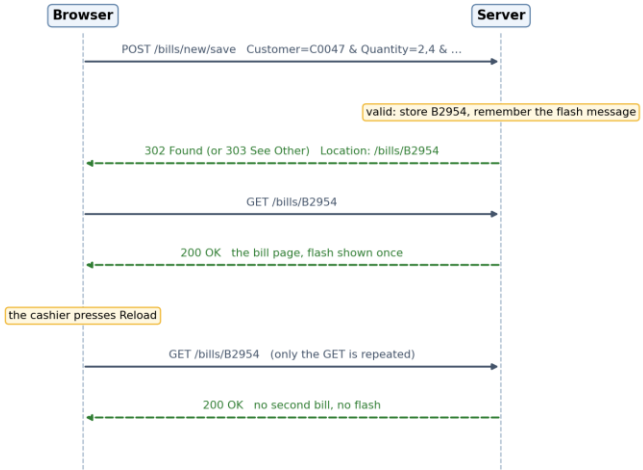


Figure 11.10 Post/Redirect/Get. The POST that stores the bill is answered by a redirect; the browser then GETs the bill page. Reload repeats only that GET, so the bill is never created twice.

***Figure 11.10**: the POST that stores the bill is answered by a redirect (302 from `RedirectToPage`, 303 in the other tracks); Reload repeats only the harmless GET. The one-time message travels in `TempData`. Book: Section 11.8.6.*

The Antiforgery Token

Listing 11.14 A POST to `/bills/new/save` without the token. The framework answers 400 before the page model is even created.

```
HTTP/1.1 400 Bad Request
Content-Length: 0
Date: Wed, 23 Sep 2026 15:15:28 GMT
Server: Kestrel
```

Every `<form method="post">` receives a hidden `__RequestVerificationToken`, paired with a cookie; with `app.UseAntiforgery()` a POST without a matching pair is refused.

Nothing had to be written to get this — the value of a framework default.

***Listing 11.14**: 400, before the page model is even created. The PHP, Java and Python tracks show what the default does for you. Book: Section 11.8.7.*

The Price Rule of the Laundry

Units, rounded up (Equation (11.3)), and price (Equation (11.4)); express is its own service, S0002, so its price splits as Equation (11.5):

$$u = \lceil w \rceil \tag{11.3}$$

$$P = p_s \cdot u \tag{11.4}$$

$$P_{\text{express}} = p_{\text{S0002}}\, u = p_{\text{S0001}}\, u + (p_{\text{S0002}} - p_{\text{S0001}})\, u \tag{11.5}$$

*The weight is rounded **up** to a whole kilogram; for items and pairs u must already be whole. The price is computed by the server from its own price list, never taken from the form. Book: Section 11.9.1.*

The Surcharge, Worked

$$s = \frac{p_{\text{S0002}} - p_{\text{S0001}}}{p_{\text{S0001}}} \times 100\% = \frac{12000 - 8000}{8000} \times 100\% = 50\% \tag{11.6}$$

> **WORKED EXAMPLE.** Imam Pratama (C0047) brings 2.4 kg for express: $u = \lceil 2.4 \rceil = 3$; regular $8000 \times 3 = 24000$; express $12000 \times 3 = 36000$; surcharge 12000, 50 %. Stored as S0002, 3 units, Rp 36.000. Curtains of 2.1 kg (S0010): $15000 \times 3 = 45000$. 2.5 shirts (per item): refused.

Equation (11.6) is computed from the price list, not written into the program, so the quote follows the owner's price changes. Book: Section 11.9.1.

Rounding Without Floating Point

In binary floating point $0.1 + 0.2 = 0.30000000000000004$: a weight a hair above a whole number would be rounded a kilogram too far.

So the text becomes whole **tenths** t (`2,4` is $t = 24$), and in integer division:

$$u = \lfloor \frac{t + 9}{10} \rfloor \tag{11.7}$$

Listing 11.15 `Tenths` in `BillRules.cs`: one to three digits and an optional single decimal, with a dot or a comma; anything else is `null`.

```
    /// <summary>"2,4" or "2.4" -> 24 tenths; null when the text is not a number of that shape.</summary>
    public static int? Tenths(string text)
    {
        var m = Regex.Match(text, @"^(\d{1,3})(?:[.,](\d))?$");
        if (!m.Success) return null;
        return int.Parse(m.Groups[1].Value) * 10 + (m.Groups[2].Success ? int.Parse(m.Groups[2].Value) : 0);
    }
```

*Equation (11.7) is Equation (11.3) computed exactly: $t = 24$ gives 3, $t = 30$ gives 3, $t = 31$ gives 4. **Listing 11.15** also stops `1e3`, `-2`, `2.45` and the empty string. Book: Section 11.9.1.*

The Server’s Answer to an Impossible Bill

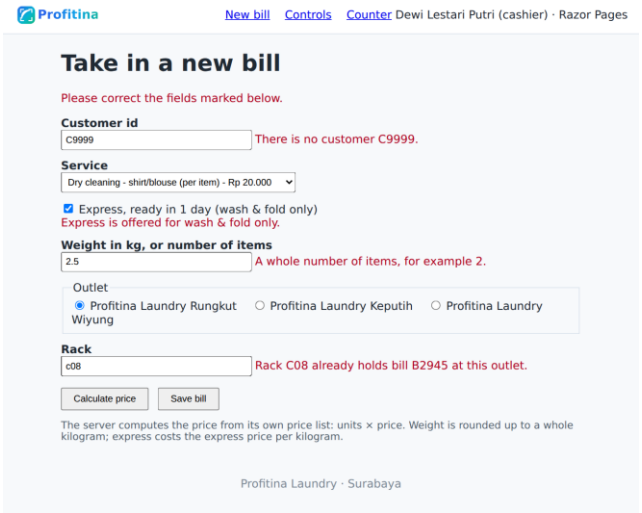


Figure 11.11 The server’s answer to an impossible bill. Four fields have errors, each next to its field, with a summary on top; every value typed is still in place, including the lowercase `c08`.

***Figure 11.11**: with the browser’s checks switched off, C9999, a shirt, express, 2.5 and rack `c08` gave four messages, each beside its field — none of which the browser alone could have checked. Every typed value stayed in place. Book: Section 11.9.3.*

Uploading a Receipt Photo

Listing 11.16 From `OnPostReceiptAsync` in `Details.cshtml.cs`: a file, at most 2 MB, JPEG or PNG by its first bytes; then a name the server makes up.

```
        string? ext = null;
        if (photo is null || photo.Length == 0) Error = "Choose a photo to upload.";
        else if (photo.Length > MaxPhoto) Error = "The photo is larger than 2 MB.";
        else if (read >= 3 && head[0] == 0xFF && head[1] == 0xD8 && head[2] == 0xFF) ext = "jpg";
        else if (read == 8 && head.SequenceEqual(new byte[] { 0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A, 0x1A, 0x0A }))
ext = "png";
        else Error = "Only JPEG or PNG photos are accepted.";      // trust the bytes, not the name
        if (Error is not null) { Response.StatusCode = 400; return Page(); }

        Directory.CreateDirectory(Uploads);
        string name = $"{id}-
{Convert.ToHexString(System.Security.Cryptography.RandomNumberGenerator.GetBytes(8)).ToLowerInvariant()}.{ext}";
        await using (var target = System.IO.File.Create(Path.Combine(Uploads, name)))   // our name, never theirs
            await photo!.CopyToAsync(target);
```

Listing 11.16*: every JPEG starts* `FF D8 FF`*, every PNG* `89 50 4E 47 0D 0A 1A 0A`*; the name and the content type are ignored. Photos are stored outside* `wwwroot` *and served by one endpoint with* `nosniff`*. Book: Section 11.9.4.*

The Bill After the Upload

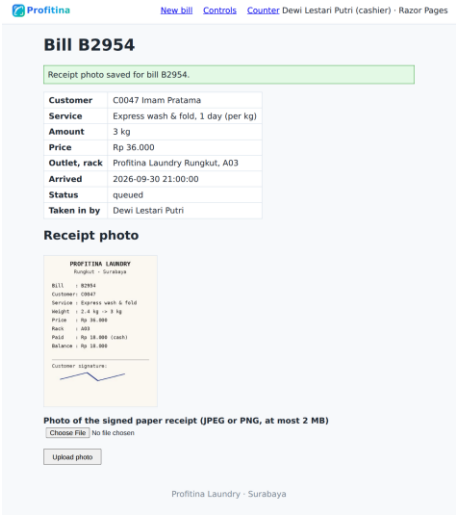


Figure 11.12 Bill B2954 after the upload: the one-time flash message, the bill as stored — S0002, 3 kg, Rp 36.000 — and the receipt photo served through `/receipts/`.

***Figure 11.12**: the one-time flash, the bill as stored — S0002, 3 kg, Rp 36.000 — and the photo served through `/receipts/`. The Razor track ran on the CSV store; the SQLite store compiles with `UseSqlite=true`. Book: Sections 11.9.4 and 11.9.5.*

The Same Form in PHP: the Token by Hand

Listing 11.17 `csrf_token` and `require_csrf` in `web.php`: 16 random bytes per session, compared with `hash_equals` before any POST handler.

```
function csrf_token(): string
{
    return $_SESSION['csrf'] ??= bin2hex(random_bytes(16));
}

/** Every POST must carry the token this session was given; anything else is refused. */
function require_csrf(): void
{
    $sent = $_POST['csrf_token'] ?? '';
    if (!is_string($sent) || !hash_equals($_SESSION['csrf'] ?? '', $sent)) {
        http_response_code(400);
        exit('The form expired or did not come from this site. Reload the page and try again.');
    }
}
```

***Listing 11.17**: PHP has no antiforgery middleware, so the track writes one. `hash_equals` takes the same time wherever the strings differ. Every view prints through `h()`: one `<?= $x ?>` without it is an XSS hole. Book: Section 11.10.*

The Same Form in Flask

Listing 11.18 `post_bill` in `python/app.py`: check, then save and redirect with 303, or quote, or redraw with 400.

```
@app.post("/bills/new")
def post_bill():
    con = laundry.connect()
    form, errors = laundry.check(request.form, con)
    quote = None
    if not errors:
        if request.form.get("action") == "save":
            bill_id = laundry.save(form, con)
            _, _, text = laundry.price(form, con)
            session["flash"] = f"Bill {bill_id} saved: {text}."
            return redirect(f"/bills/{bill_id}", code=303)         # Post/Redirect/Get
        quote = laundry.price(form, con)[2]                           # "Calculate price" only
    return render_template("new_bill.html", form=form, errors=errors, quote=quote,
                           services=laundry.services(con), outlets=laundry.outlets(con)), \
        (400 if errors else 200)
```

***Listing 11.18**: `before_request` checks the token with `hmac.compare_digest`; `MAX_CONTENT_LENGTH` refuses over 3 MB with 413; Jinja2 escapes every value. The Java track parses urlencoded and multipart bodies by hand. Book: Sections 11.11 and 11.12.*

Cross-Site Request Forgery

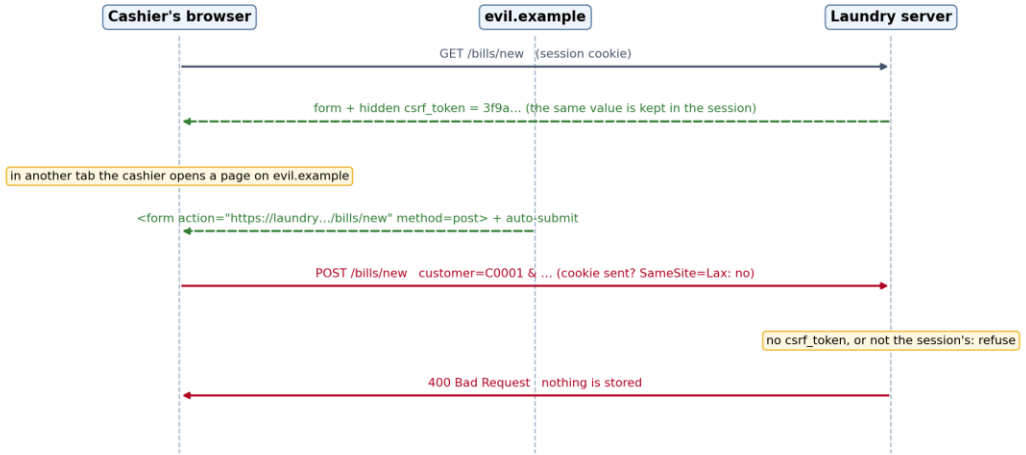


Figure 11.13 A cross-site request forgery and the two defenses. A page on `evil.example` makes the cashier's browser post to the Laundry; the post is refused because it does not carry the session's token, and with `SameSite=Lax` the session cookie is not even sent.

***Figure 11.13**: a page on `evil.example` makes the cashier's browser post to the Laundry with her cookie. The synchronizer token stops it — the attacker can neither read nor guess it — and `SameSite=Lax` keeps the cookie off the forged request. Book: Section 11.13.3.*

Guessing the Token

A token of 16 random bytes is 128 bits; one guess succeeds with probability Equation (11.8):

$$\Pr[\text{guess}] = 2^{-128} \approx 2.9 \times 10^{-39} \tag{11.8}$$

Validate on the server — shape, range, meaning, consistency; never accept the price from the form.

Escape on the way out — `&`, `<`, `>`, `"` and `'`; an unescaped `"` lets `" autofocus onfocus="alert(1)` add attributes.

At a billion guesses a second an attacker would need far longer than the age of the universe. `SameSite` *is a second wall, not a replacement. Book: Sections 11.13.1–11.13.3.*

Uploads From Strangers

Table 11.8 What can go wrong with an upload, and what the Laundry code does. Checking the first bytes is not a malware scan; it only ensures the file is the kind of file it claims to be.

Risk	Example	Defense in all four tracks
the name is a path	`../../web.config`	never use the sent name; the server makes one up
the name is markup	`<img src=x onerror=…>.png` shown on a page	never use the sent name; escape anything shown
the type is a lie	an HTML page sent as `image/png`	check the first bytes: `FF D8 FF` or `89 50 4E 47 …`
the file is served as a page	`receipt.html` served from the web root, running script	store outside the web root; serve through one handler with a fixed type and `nosniff`
the file is huge	2 GB, or a million small requests	a request limit at the server and a smaller one in the page
guessable names	`B2954-1.png`, `B2954-2.png`, …	64 random bits in every name
malware	a real JPEG carrying something else	out of scope here; scan uploads in production
personal data	a signed receipt shows a name and a signature	keep only what is needed; control who can view it

***Table 11.8**: checking the first bytes is not a malware scan. A photo of a signed receipt is personal data under UU No. 27 of 2022: keep it for a reason, keep it safe, not longer than needed. Book: Section 11.13.4.*

Twenty Checks, Four Servers

Table 11.9 The same form in four technologies, first part: routing, parsing, the pressed button, the token and escaping.

Job	Razor Pages	PHP	Java HttpServer	Flask
routing	`@page` templates	`index.php` + `router.php`	one context, `if` chain	`@app.get`, `@app.post`
parsing urlencoded	model binding	`$_POST`	`Forms.urlencoded` (by hand)	`request.form`
parsing multipart	`IFormFile`	`$_FILES`	`Forms.multipart` (by hand)	`request.files`
which button	`asp-page-handler`, `formaction`	`name="action"`	`name="action"`	`name="action"`
CSRF token	automatic, `__RequestVerificationToken`	`csrf_token()` + `hash_equals`	session map + `MessageDigest.isEqual`	`before_request` + `compare_digest`
escaping	automatic	`h()` = `htmlspecialchars`	`h()` by hand	automatic (Jinja2)

***Table 11.9**: every job is done in every track; what changes is how much the framework does for you. `tests/form_suite.py` ran the same twenty requests against each: 20 passed, 0 failed, four times. Book: Section 11.13.5.*

Four Tracks, Continued

Table 11.10 The same form in four technologies, second part: the one-time message, the redirect, the size limit, the data and the port.

Job	Razor Pages	PHP	Java HttpServer	Flask
one-time message	`TempData`	`$_SESSION`	session map	`session.pop`
redirect after POST	302 `RedirectToPage`	303 `header`	303 `sendResponseHeaders`	303 `redirect`
size limit	`[RequestSizeLimit]`	`php.ini` / `-d`	`MAX_BODY`	`MAX_CONTENT_LENGTH`
data	CSV in memory (SQLite with one switch)	PDO, SQLite	JDBC, SQLite	`sqlite3`
port in this chapter	9101	9102	9103	9104

***Table 11.10**: Web Forms did the most and hid the most; the JDK HttpServer does nothing and hides nothing. Know what each row involves, especially when the framework does it. Book: Section 11.13.5.*

Profitina in Practice

The server decides: what a scan costs and which tier allows it come from the account, never from the request; FastAPI binds and validates into typed models.

Consent that means something: an unchecked box sends nothing, so a missing value is *no*; deletion, export and password change are protected forms.

Uploads and text from strangers: type by bytes, name by server, output escaped; posted links screened by Link Safety.

The runnable case study is always Profitina Laundry. What carries across: input checked on the server, a price computed, output escaped, every state change carrying proof of origin. Book: Section 11.14.

Common Mistakes

Forms: a file input without `multipart/form-data`; reading a missing checkbox as an error; trusting `required` and `pattern`.

Trust: taking the price or role from the form; binding a whole entity; using the upload's name or content type.

Flow: answering a successful POST with a page; a POST without a CSRF token; printing a typed value unescaped; parsing `2,4` with the server's culture.

Web Forms and Razor: refilling a list without `IsPostBack`; secrets in ViewState; changing a bound property without `ModelState.Remove`.

Each mistake is listed in the book with its cure. Book: Section 11.15.

Summary: Five Things to Remember

1. Static pages are files, dynamic pages are programs; read-change-write needs protection.

2. A form sends text by strict rules: named, enabled, checked, pressed; GET in the URL, POST in the body; three encodings.

3. Web Forms hid HTTP with server controls, postback and ViewState; Razor Pages keeps the ideas and drops the hiding.

4. The server prices, validates and redirects: Equation (11.7), Post/Redirect/Get, one antiforgery token per form.

5. Four habits: validate on the server, escape on output, a token on every state-changing POST, uploads treated as hostile.

Book: Section 11.16.

Next Time

Chapter 12 — Evaluation and Discussion III. Server pages, forms and state from JSP to Razor Pages: problems on Chapters 9 to 11, worked in full.

Before then: work Exercises 11.1 to 11.4 and check each against a running track.

The solutions to the exercises are in the book. Book: Sections 11.17 and 11.18.

Lecture 12 — Evaluation and Discussion III: Server Pages, Forms and State, from JSP to Razor Pages

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 12

Lecturer: Irfan Subakti — yifana@gmail.com

No new technology today: eleven problems, each solved from a blank page and each run on a real machine, on the ideas of Chapters 9 to 11 and the JSP of Chapter 7. Book: Chapter 12, Sections 12.1–12.12.

What This Meeting Covers

1. **A method** — six questions to ask of any problem about a page, a form and some state, before any code.
2. **The project (Problems 12.1–12.4)** — a JSP website with eight CRUD features: data model, features, a measurable design, evidence and delivery.
3. **Pages and state (12.5, 12.6)** — a Web Forms page as a Razor page; a visitor counter that does not lose visitors.
4. **Forms and handlers (12.7, 12.8)** — rules that hold whatever the browser does; one page, two buttons.
5. **Rich clients, five systems, forged forms (12.9–12.11)**, then the problems compared.

Each problem follows one order: the problem, in our own words, design, the solution, Java and Python, line by line, a tempting wrong solution, check yourself. Book: Section 12.1.

Where We Left Off: the Toolbox

Table 12.1 The tools of Chapters 7 and 9 to 11, and the problems of this chapter that use them.

Chapter	What it gave us	Where this chapter uses it
7 (recalled)	JSP pages, directives, scriptlets, the implicit objects, EL, JSTL, the four scopes	Problems 12.1 to 12.4, 12.10
9	ASP.NET: the page syntax of Web Forms (`<%@ Page %>`, `<script runat="server">`, `<%= %>`), server controls, validation controls; ASP.NET Core Razor Pages; Rich Internet Applications and AJAX, thin and fat clients	Problems 12.5, 12.9
10	Three systems side by side: PHP on Apache, JSP on Tomcat, ASP.NET on IIS or Kestrel; installing and running each; the Razor Pages project and its files	Problems 12.4, 12.10
11	Static and dynamic pages (the visitor counter), single-file and code-behind page classes, web forms, web controls, events and their handlers, application and session state	Problems 12.6, 12.7, 12.8, 12.11

***Table 12.1**: two threads run through all four chapters — **a page on a server is a program**, and **the web forgets**: nothing survives between requests unless you put it somewhere on purpose. Book: Section 12.1.*

Six Questions, in Order

1. **What are the requests?** Method, path, parameters; anything that changes data is a `POST`.
2. **Where does each piece of state live?** Database, session, application, request, or the page itself.
3. **Who may do what?** Checked on the server for every request; hiding a button is a courtesy, never a protection.
4. **What must the server check?** Every value from a browser is a claim, not a fact. **5. How is output escaped?** `<c:out>`, `@`, `{{ }}`, `h()`. **6. How will you prove it works?**

> **THE ORDER MATTERS.** Requests, state, permissions, checks, printing, proof — and only then the page. A team that starts with the page moves state around on the last evening.

On a small problem this takes five minutes; on a large one it is the design. Book: Section 12.2.

The Life of One Form

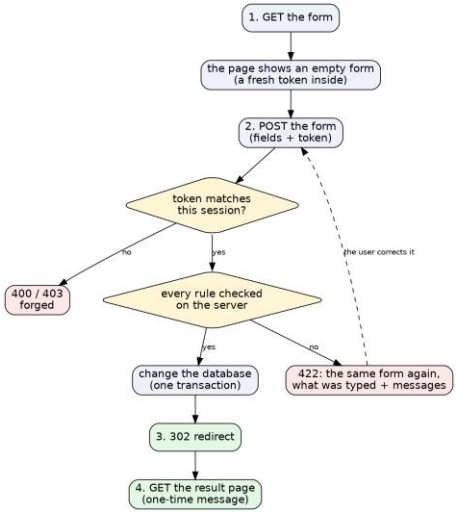


Figure 12.1 The life of one form. Only a valid form with a valid token reaches the database, and a successful change always ends in a redirect.

***Figure 12.1**: a `GET` shows the form with a fresh token; a wrong token ends the request; a wrong field shows the form again with what was typed; only a valid form changes the database, followed by a redirect — Post/Redirect/Get. Book: Section 12.2.*

The Case and the Code

Profitina Laundry: three outlets, fourteen employees, twelve services, 256 customers, 2,953 bills, one year of trading ending on 30 September 2026.

Tomcat could not be downloaded, so the JSP pages ran on **MiniJSP**, a small JSP engine that translates each page into a servlet class and serves it with the JDK's HTTP server.

Every server has its own port from 9200 to 9299; the Razor Pages read the CSV export; Web Forms listings are for reading only.

`Code_new/Chapter12` *holds* `jsp/`, `minijsp/`, `java/`, `python/`, `aspnet/LaundryPages/`, `webforms/`, `php/`, `design/`, `tests/` *and* `deploy/`; `VERIFIED.md` *records exactly what was run. Book: Section 12.3.*

The Project: Four Problems

The brief: a JSP website with at least eight CRUD features, made as beautiful as possible, with a write-up (data model, features, screenshots), a free host and a public repository.

12.1 the data model · **12.2** the eight features · **12.3** “beautiful”, made measurable · **12.4** evidence and delivery.

The Laundry Back Office, the owner’s side: four features for services (list, add, edit, delete) and four for outlets (overview, open, edit, close).

Every letter of CRUD appears twice, and the two deletes are exactly the ones a database must sometimes refuse. Login is needed but not counted among the eight. Book: Section 12.4.

Problem 12.1: Conceptual and Physical

> **DEFINITION.** An *entity* is a thing the business keeps facts about; a *relationship* connects two entities, and its *cardinality* says how many of one go with one of the other (1 : N). Physically an entity becomes a *table*, a 1 : N relationship a *foreign key* on the N side.

Underline the nouns of the features: a noun with facts of its own is an entity (Service, Outlet, Bill, Employee); a single fact is an attribute; *queue* is only a question asked of Bill.

The conceptual model is what a shop owner recognizes; the physical model is what the database really contains — and it must be true. Book: Section 12.4.1.

The Conceptual Model

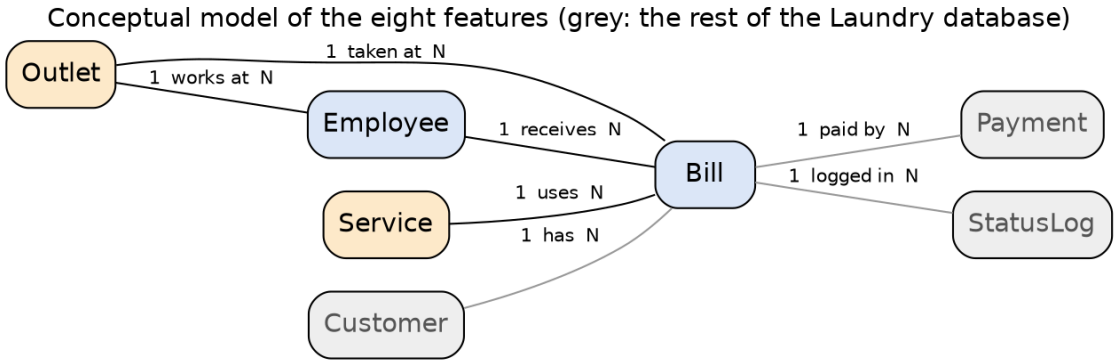


Figure 12.2 The conceptual model of the eight features. Outlet and Service are the entities the features change; every line is one-to-many.

***Figure 12.2**: Outlet and Service sit on the "1" side of relationships with Bill, so features F4 and F8 must turn the database's refusal into a sentence. A bill stores what it charged, so F3 may change a price freely. Book: Section 12.4.1.*

The Physical Model, From the Database

Listing 12.1 `schema_report.py`: the physical model as Markdown, from SQLite's own `PRAGMA table_info` and `PRAGMA foreign_key_list`.

```
DB = os.environ.get("LAUNDRY_DB", os.path.join(os.path.dirname(__file__), "..", "data", "laundry.db"))
con = sqlite3.connect(DB)
wanted = set(sys.argv[1:])
tables = [r[0] for r in con.execute(
    "SELECT name FROM sqlite_master WHERE type = 'table' AND name NOT LIKE 'sqlite_%' ORDER BY name")]
for t in tables:
    if wanted and t not in wanted:
        continue
    fks = {r[3]: f"{r[2]}.{r[4]}" for r in con.execute(f"PRAGMA foreign_key_list({t})")}  # from -> table.to
    count = con.execute(f"SELECT COUNT(*) FROM {t}").fetchone()[0]                        # t is a real table name
    print(f"### {t} ({count} rows)")
    print("| Column | Type | Null? | Key |")
    print("|:--|:--|:--|:--|")
    for cid, name, ctype, notnull, default, pk in con.execute(f"PRAGMA table_info({t})"):
        key = "PK" if pk else f"FK to {fks[name]}" if name in fks else ""
        print(f"| {name} | {ctype} | {'NOT NULL' if notnull or pk else 'NULL'} | {key} |")
    print()
```

***Listing 12.1**: the write-up cannot disagree with the tables. Tempting and wrong: a hand-drawn table, or foreign keys not enforced. Book: Section 12.4.1.*

Problem 12.2: Eight CRUD Features

Table 12.2 The eight features of the Laundry Back Office. Every one is for the logged-in owner, and every change is a POST.

#	Feature	Requests (JSP version)	CRUD
F1	Price list with search and how often each service was ordered	`GET services.jsp?q=`	R
F2	Add a service	`GET`, `POST service-form.jsp`	C
F3	Edit a service's description or price	`GET`, `POST service-form.jsp?id=S0013`	U
F4	Delete a service (refused while bills use it)	`POST service-delete.jsp`	D
F5	Outlets overview, with staff and queue	`GET outlets.jsp`	R
F6	Open an outlet	`GET`, `POST outlet-form.jsp`	C
F7	Edit an outlet	`GET`, `POST outlet-form.jsp?id=O04`	U
F8	Close an outlet (refused while employees or bills belong to it)	`POST outlet-delete.jsp`	D

***Table 12.2**: services and outlets live in the database; who is logged in and the form token in the session; the rows a page prints in the request; the one-time message in the session, across the redirect. Book: Section 12.4.2.*

One Model, Three Front Ends

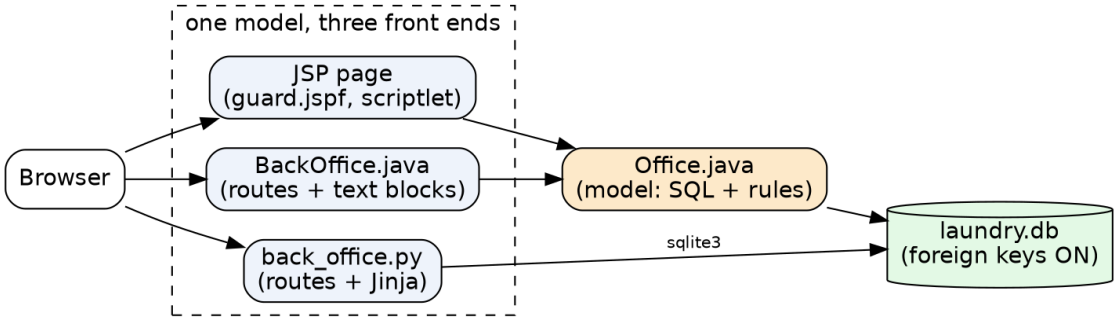


Figure 12.3 One model, three front ends. The JSP pages and the JDK server share `Office.java`; the Flask version mirrors it with `sqlite3`.

***Figure 12.3**: every SQL statement and every rule lives in one plain Java class, `Office`, that knows nothing about HTML; the pages hold only a guard, a call and the HTML that prints the result. Book: Section 12.4.2.*

The Guard, Pasted Into Every Page

Listing 12.2 `guard.jspf`: no owner, no page; no token, no change; and the page's model object.

```
<%-- guard.jspf - pasted at the top of every page of the back office except login.jsp.
     1. Nobody but the logged-in owner goes further.
     2. A POST must carry the form token kept in the session (no forged forms).
     3. Gives the page its model object, "office". --%>
<%
    if (session.getAttribute("owner") == null) {
        response.sendRedirect("login.jsp");
        return;
    }
    if ("POST".equals(request.getMethod())
            && !Office.sameToken(session.getAttribute("csrf"), request.getParameter("csrf"))) {
        response.setStatus(403);
        out.print("Forbidden: this form has expired. Go back, reload the page and try again.");
        return;
    }
    Office office = Office.of(application.getInitParameter("laundry.db.url"));
%>
```

***Listing 12.2**: included with `<%@ include file="…" %>` from `WEB-INF`, where no browser can ask for it. `return;` ends the page's service method at once — legal because the page is a method. The token is compared in constant time. Book: Section 12.4.2.*

F1: the Price List

Listing 12.3 The rows of `services.jsp` (F1): every value through `<c:out>`, and Delete as a POST form carrying the token.

```
<c:forEach var="s" items="${services}">
  <tr>
    <td><c:out value="${s.id}"/></td>
    <td><c:out value="${s.description}"/></td>
    <td class="num"><c:out value="${s.priceText}"/></td>
    <td class="num"><c:out value="${s.bills}"/></td>
    <td class="actions">
      <a href="service-form.jsp?id=<c:out value='${s.id}'/>">Edit</a>
      <form method="post" action="service-delete.jsp" class="inline"
            onsubmit="return confirm('Delete this service?')">
        <input type="hidden" name="csrf" value="${sessionScope.csrf}">
        <input type="hidden" name="id" value="<c:out value='${s.id}'/>">
        <button class="danger">Delete</button>
      </form>
    </td>
  </tr>
</c:forEach>
```

***Listing 12.3**: the page's scriptlet is one line — ask the model for the services matching `q` and put them in request scope; the search text reaches SQL only as a value for a `?` placeholder. Book: Section 12.4.2.*

F4: Delete, POST Only

Listing 12.4 `service-delete.jsp` (F4): POST only; the foreign key decides, the page reports.

```
<%@ page contentType="text/html; charset=UTF-8" import="laundry.office.*" %>
<%-- service-delete.jsp - F4: POST only; the database refuses while bills use the service. --%>
<%@ include file="/WEB-INF/jspf/guard.jspf" %>
<%
    if (!"POST".equals(request.getMethod())) {
        response.setStatus(405);
        out.print("Use the Delete button on the price list.");
        return;
    }
    String id = request.getParameter("id");
    String error = office.deleteService(id);
    if (error == null) session.setAttribute("flash", "Service " + id + " deleted.");
    else session.setAttribute("flashError", error);
    response.sendRedirect("services.jsp");
%>
```

> **The tempting wrong solution: a delete link.** `<a href="service-delete.jsp?id=S0005">` also “clicks” for a prefetching browser, a chat preview, or `<img src=…>` on another site. `GET` must be *safe*.

Listing 12.4 *refuses a `GET` with 405, lets the foreign key decide, and reports the verdict after a redirect. Book: Section 12.4.2.*

Problem 12.3: Beauty, Made Checkable

1. Few colors, each a named custom property (`--brand: #0b5cad`). **2.** Every text color readable: contrast at least 4.5 : 1 (3 : 1 for large text).

3. One spacing step, `--step`. **4.** One type scale, tabular digits for rupiah.

5. Visible focus with `:focus-visible`. **6.** Nothing breaks on a phone: below 40rem each table row becomes a card.

Beauty cannot be proved, but most of what makes a site look cared for can be — and the rest can be made consistent. One stylesheet serves all versions byte for byte. Book: Section 12.4.3.

Contrast, From the Definition

Each channel $c_s = c/255$ is made linear by Equation (12.1); relative luminance weighs the linear channels by Equation (12.2); the ratio of the lighter L_1 to the darker L_2 is Equation (12.3):

$$c_\ell = \begin{cases} c_s/12.92 & \text{if } c_s \le 0.04045 \\ \left(\frac{c_s+0.055}{1.055}\right)^{2.4} & \text{otherwise.} \end{cases} \tag{12.1}$$

$$L = 0.2126\,R_\ell + 0.7152\,G_\ell + 0.0722\,B_\ell \tag{12.2}$$

$$\text{ratio} = \frac{L_1 + 0.05}{L_2 + 0.05}, \tag{12.3}$$

The eye is most sensitive to green and least to blue; the 0.05 *stands for the light a real screen reflects even when it shows black. The ratio runs from 1 to 21. Book: Section 12.4.3.*

The Secondary Text, Worked by Hand

> **WORKED EXAMPLE.** `--muted` is `#52606d`: 82, 96, 109, scaled to 0.3216, 0.3765, 0.4275. Linear: 0.0844, 0.1170, 0.1529. $L = 0.2126 \times 0.0844 + 0.7152 \times 0.1170 + 0.0722 \times 0.1529 = 0.1126$. Against white: $(1 + 0.05)/(0.1126 + 0.05) = 6.46$, above 4.5.

Listing 12.5 The two functions of `contrast.py`: luminance by Equation (12.1) and Equation (12.2), and the ratio by Equation (12.3).

```
def luminance(hex_color):
    """Relative luminance of an sRGB color such as #0b5cad: 0 for black, 1 for white."""
    channels = [int(hex_color[i:i + 2], 16) / 255 for i in (1, 3, 5)]
    r, g, b = [c / 12.92 if c <= 0.04045 else ((c + 0.055) / 1.055) ** 2.4 for c in channels]
    return 0.2126 * r + 0.7152 * g + 0.0722 * b


def ratio(fg, bg):
    lighter, darker = sorted((luminance(fg), luminance(bg)), reverse=True)
    return (lighter + 0.05) / (darker + 0.05)
```

Listing 12.5 *prints every pair of the stylesheet; the tempting soft grey* `#9aa5b1` *on white comes out at 2.50 : 1 — a failure even for large text. Book: Section 12.4.3.*

F1 as Served

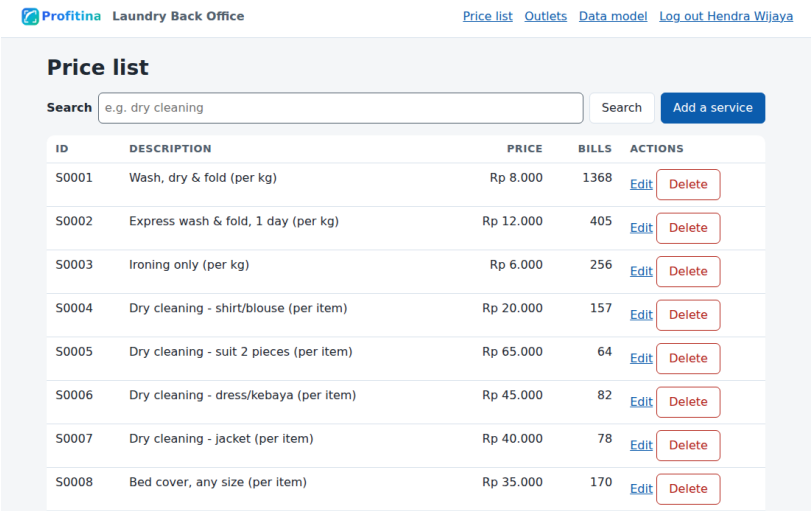


Figure 12.4 Feature F1, the price list, served by the JSP pages on MiniJSP. Numbers are right-aligned; Delete is red and asks for confirmation.

Figure 12.4 *was taken by a program, in headless Chromium, from the running site: numbers right-aligned, Delete red and asking for confirmation. Book: Section 12.4.3.*

Problem 12.4: an Acceptance Test

```
PASS A1 guard: not logged in -> 302 login.jsp
PASS A2 wrong password: 401, message shown
PASS A3 cashier refused: a cashier's correct password -> 401
PASS A4 owner login: 302 services.jsp, new session id: True
PASS F1 list: 12 services, S0005 costs Rp 65.000
PASS F1 search: 'dry clean' -> S0004 S0005 S0006 S0007
PASS F2 refused: 422 The price must be from Rp 1 to Rp 99.999.
PASS F2 add: 302, then: Service added. S0013 listed
PASS F3 edit: 302, S0013 now Rp 50.000
PASS F4 refused: Service S0001 appears on bills and cannot be deleted.
PASS F4 delete: Service S0013 deleted.
PASS F5 outlets: 3 outlets, queues 9/9/5
```

The test talks HTTP, keeps cookies, never follows a redirect, and finds its way by a few marks all three versions print; the run ends *21 of 21 checks passed*.

The first lines of the run against the JSP pages, on a fresh copy of the database. Screenshots show one day; this proves it again in a second after every change. Book: Section 12.4.4.

A Refusal, Photographed by a Program

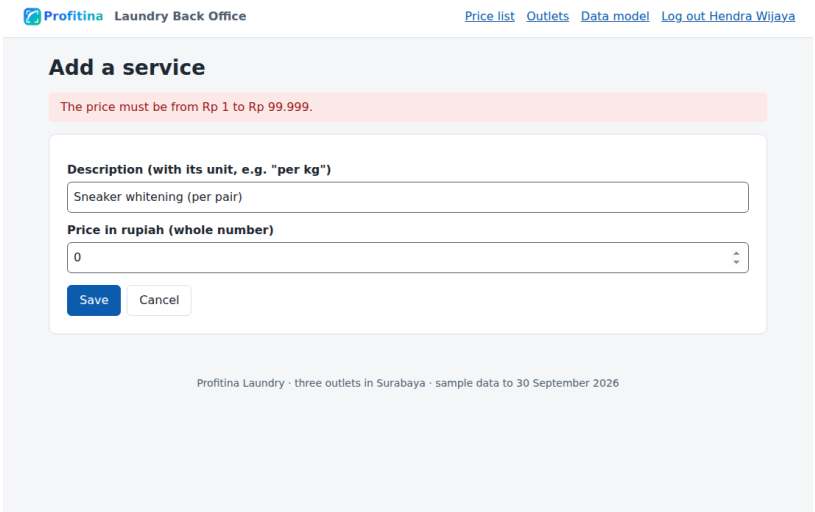


Figure 12.5 Feature F2 refusing a price of 0: status 422, the message at the top, and what was typed still in the fields.

***Figure 12.5**: F2 refuses a price of 0 with status 422, the message at the top and what was typed still in the fields. `tests/screenshots.py` drives Chromium with Playwright. Book: Section 12.4.4.*

The Write-Up, Assembled From Programs

Table 12.3 The parts of the write-up and where each comes from.

Part	What it contains	Where it comes from
Database design	conceptual model; physical model	**Figure 12.2**; `SchemaReport` or `schema_report.py` output
Features	one row per feature: what, who, requests, CRUD	**Table 12.2**
Evidence	one screenshot and two or three sentences per feature; the acceptance run	`screenshots.py`; `acceptance.py` output
Design	the six rules; the contrast table	Problem 12.3; `contrast.py` output
How to run and deploy	commands, settings, the address of the live site	`README.md`; `build-war.sh`, `Dockerfile`
Team	each member's deliverables and share	a table like the one described below

***Table 12.3**: nothing is typed from memory when a program can print it. For a team, give each member's concrete deliverables and share; a count of commits is not a measure of work. Book: Section 12.4.4.*

Packaging, Hosting, Repository

A WAR for Tomcat 10.1 or newer: pages at the top, classes in `WEB-INF/classes`, libraries in `WEB-INF/lib`; Tomcat compiles the JSP pages and needs the two JSTL jars.

A free host that runs *containers*: most free hosts run only PHP. A container's disk is usually wiped on redeploy, so real data belongs in a hosted database.

The repository: code, never data or secrets — `.gitignore` keeps `laundry.db`, `.env` and keys out; a pushed secret must be changed.

The WAR build ran; Tomcat and Docker were not available, so the deployment steps are templates. Committing `laundry.db` would publish every customer's name and phone. Book: Section 12.4.4.

Problem 12.5: Web Forms to Razor Pages

Table 12.4 Each part of a Web Forms page and its Razor Pages counterpart: page, code, first visit, button, text box and list.

Web Forms (`.aspx`)	Razor Pages (`.cshtml` + `.cshtml.cs`)
`<%@ Page %>`, `CodeFile=`, `Inherits=`	`@page`, `@model QuoteModel`
`<script runat="server">` or the code-behind class	the page model class, `QuoteModel : PageModel`
`Page_Load` with `if (!IsPostBack)`	`OnGet()` (first visit) and `OnPost()` (the form came back)
`QuoteButton_Click`	`OnPost()`, or a named handler `OnPostQuote()` with `asp-page-handler="Quote"`
`<asp:TextBox ID="UnitsBox" runat="server">`	`<input asp-for="Units">` bound to `[BindProperty] public int Units`
`<asp:DropDownList>` filled in `Page_Load`	`<select asp-for="ServiceId" asp-items="…">`

***Table 12.4**: Razor Pages keeps the pair of files and a handler per request, and drops ViewState. Book: Section 12.5.1.*

Web Forms to Razor Pages, Continued

Table 12.5 Each part of a Web Forms page and its Razor Pages counterpart: validators, the label, ViewState and the render block.

Web Forms (`.aspx`)	Razor Pages (`.cshtml` + `.cshtml.cs`)
`<asp:RangeValidator>`, `<asp:RequiredFieldValidator>`	`[Range]`, `[Required]` on the property; `<span asp-validation-for>`
`Page.IsValid`	`ModelState.IsValid`
`ResultLabel.Text = …`	a property `Result`, printed with `@Model.Result`
`__VIEWSTATE` hidden field	nothing: state is rebuilt from the database or the posted fields
`<%= value %>`	`@value` (escaped by default)

***Table 12.5**: what disappears is `__VIEWSTATE`, and with it the illusion that a page remembers. Book: Section 12.5.1.*

The Quote, Computed on the Server

Listing 12.6 `OnPost` in `Quote.cshtml.cs`: look the service up, stop with 422 if anything is invalid, multiply on the server.

```
    public void OnPost()                    // the button was pressed: the form comes back
    {
        var service = store.Service(ServiceId);
        if (service is null) ModelState.AddModelError(nameof(ServiceId), "Choose a service from the list.");
        if (!ModelState.IsValid)
        {
            Response.StatusCode = 422;      // the page is shown again, with the messages
            return;
        }
        Result = $"{Units} x {Money.Rupiah(service!.Price)} = {Money.Rupiah(service.Price * Units)}";
    }
```

***Listing 12.6**: `Units` is a `[BindProperty]` with `[Range(1, 100)]`, so `abc` fails in model binding before any rule runs; the `Store` arrives by dependency injection. Book: Section 12.5.1.*

The Tempting Way: a Total From the Browser

```
curl -s -d "ServiceId=S0005&Units=10&Total=1" http://127.0.0.1:9221/wrong
```

```
<p id="result">10 units, total Rp 1</p>
```

Ten two-piece suits for one rupiah. A hidden field is part of the form, and the form is whatever the sender makes it.

The `/wrong` page of the Flask track believes a total computed by JavaScript in the browser. A price that comes from the browser is a price the customer chooses. Book: Section 12.5.1.

Problem 12.6: the Counter, With a Lock

Listing 12.7 From `CounterModel`: `OnGet` bumps the count inside `lock (Gate)`; `OnGetNaive` does not.

```
    public void OnGet()
    {
        lock (Gate)                                  // read-add-write happens for one request at a time
        {
            Visitor = Bump(FileFor("visitors-safe.txt"));
        }
    }

    public void OnGetNaive() => Visitor = Bump(FileFor("visitors-naive.txt"));

    private static long Bump(string file)            // inside a PageModel, File() makes a download: say
System.IO.File
    {
        long n = System.IO.File.Exists(file) && long.TryParse(System.IO.File.ReadAllText(file), out var v) ? v :
0;
        n++;                                         // two requests that read the same n both write n + 1
        System.IO.File.WriteAllText(file, n.ToString());
        return n;
    }
```

***Listing 12.7**: `Gate` must be a `static readonly object` — ASP.NET Core creates a new page model per request, so an ordinary field would lock nothing. Book: Section 12.5.2.*

Four Hundred Visitors, Sixteen at a Time

The measurement counts what the counter lost:

$$\text{lost} = \text{visitors sent} - \text{highest number shown} \tag{12.4}$$

```
http://127.0.0.1:9210/Counter?handler=Naive
  visitors sent 400, answered 400, errors 0
  highest number shown 78, distinct numbers 78
  visitors lost by the counter 322
http://127.0.0.1:9210/Counter
  visitors sent 400, answered 400, errors 0
  highest number shown 400, distinct numbers 400
  visitors lost by the counter 0
```

By Equation (12.4), the naive counter lost 322 of 400 on the Razor pages (287 in Java, 339 in Python); the safe one counted exactly 400. The naive numbers change from run to run; the safe number never does. Book: Section 12.5.2.

Why So Many Were Lost

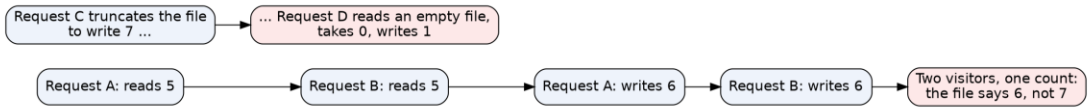


Figure 12.6 Two ways to lose visitors without a lock: two requests read the same number, or one request reads the file while another has emptied it to rewrite it.

Figure 12.6*: the classic lost update loses a handful; the hundreds come from reading the file while another request has emptied it to rewrite it, and treating "nothing" as zero. Across two servers only the database can decide:* `UPDATE Counter SET n = n + 1`*. Book: Section 12.5.2.*

Problem 12.7: Rules That Hold Whatever the Browser Does

A cashier takes in a bill: a customer id like `C0047` that exists, a service, a quantity from 1 to 100, a rack like `A07`, an outlet. Every mistake beside its field; what was typed kept.

Shape rules are declared as attributes; **existence** rules need the data and are written in the handler. The charge is computed on the server:

$$\text{charge} = \text{price of the service} \times \text{quantity}, \tag{12.5}$$

Equation (12.5) is computed from the price list at take-in and stored in the bill, so later price changes do not rewrite it. Book: Section 12.6.1.

Shape Rules as Attributes

Listing 12.8 The input class of `Bills/New.cshtml.cs`: shape rules declared as attributes, checked during model binding.

```
    public sealed class InputModel
    {
        [Required(ErrorMessage = "Give the customer id.")]
        [RegularExpression(@"^C\d{4}$", ErrorMessage = "A customer id looks like C0047.")]
        public string? CustomerId { get; set; }

        [Required(ErrorMessage = "Choose a service.")]
        public string? ServiceId { get; set; }

        [Range(1, 100, ErrorMessage = "The quantity must be a whole number from 1 to 100.")]
        public int Unit { get; set; } = 1;

        [Required(ErrorMessage = "Give the rack.")]
        [RegularExpression(@"^[A-Z]\d{2}$", ErrorMessage = "A rack looks like A07: one capital letter, two
digits.")]
        public string? Rack { get; set; }

        [Required]
        public string OutletId { get; set; } = "O01";
    }
```

***Listing 12.8**: the fields are called `Input.CustomerId` and so on; each failure is recorded in `ModelState` under its name. The Web Forms counterpart is a validator control beside each text box. Book: Section 12.6.1.*

Existence Rules in Code, the Charge on the Server

Listing 12.9 `OnPost` of `Bills/New.cshtml.cs`: existence rules added to the same `ModelState`, the charge computed by the store, then a redirect.

```
        // The attributes checked the shape of each value; only the server can check that it exists.
        if (ModelState.IsValid && store.Customer(Input.CustomerId) is null)
            ModelState.AddModelError("Input.CustomerId", $"There is no customer {Input.CustomerId}.");
        var service = store.Service(Input.ServiceId);
        if (service is null)
            ModelState.AddModelError("Input.ServiceId", "Choose a service from the list.");
        if (store.Outlets.All(o => o.Id != Input.OutletId))
            ModelState.AddModelError("Input.OutletId", "Choose an outlet from the list.");
        if (!ModelState.IsValid)
        {
            Response.StatusCode = 422;              // the same form again, with the messages
            return Page();
        }
        var now = config["LaundryNow"] ?? DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
        var bill = store.AddBill(Input.CustomerId!, service!, Input.Unit, Input.Rack!, Input.OutletId, now);
        TempData["Flash"] = $"Bill {bill.Id} created: {bill.Unit} x {Money.Rupiah(service!.Price)} =
{Money.Rupiah(bill.Charged)}.";
        return RedirectToPage("Detail", new { id = bill.Id });      // Post/Redirect/Get
```

***Listing 12.9**: the existence check runs only if the shapes were valid; after success, `TempData` carries the message across the redirect to the bill's page. Book: Section 12.6.1.*

Three Mistakes, One Answer

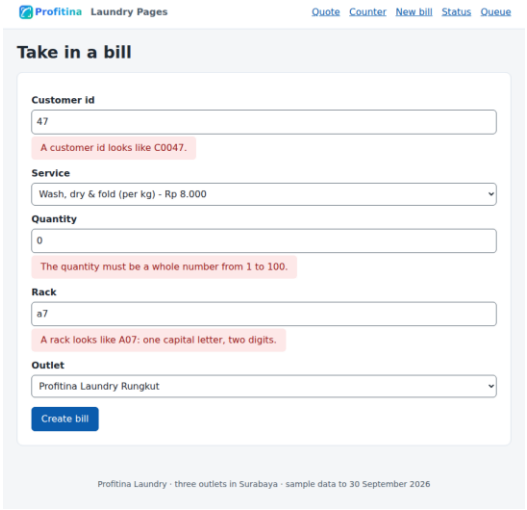


Figure 12.7 The Razor form after `47`, `0` and `a7`: three messages, status 422, nothing written.

***Figure 12.7**: `47`, `0` and `a7` gave three messages and status 422; nothing was written. Those requests were sent by a program that never ran the page's HTML — the browser's `required` and `pattern` never saw them. Book: Section 12.6.1.*

Problem 12.8: One Page, Two Buttons

Listing 12.10 `OnPostAdvance` and `OnPostPay` in `Bills/Detail.cshtml.cs`: each asks the store, leaves a message in `TempData`, and redirects.

```
    public IActionResult OnPostAdvance(string id)
    {
        var error = store.Advance(id, Now);
        if (error is null) TempData["Flash"] = $"Bill {id} is now {store.Bill(id)!.Status}.";
        else TempData["Error"] = error;
        return RedirectToPage(new { id });          // back to OnGet: reloading repeats nothing
    }

    public IActionResult OnPostPay(string id, long amount, string? method)
    {
        var error = store.Pay(id, amount, method, Now);
        if (error is null) TempData["Flash"] = $"Payment of {Money.Rupiah(amount)} recorded.";
        else TempData["Error"] = error;
        return RedirectToPage(new { id });
    }
```

***Listing 12.10**: `asp-page-handler="Advance"` writes `?handler=Advance`; the rules (no moving a collected bill, no payment above the balance) live in the store. Answering a POST with a page would let Reload record a payment twice. Book: Section 12.6.2.*

The Cashier, Played by a Script

```
1. three mistakes      -> 422
     A customer id looks like C0047.
     The quantity must be a whole number from 1 to 100.
     A rack looks like A07: one capital letter, two digits.
2. unknown customer    -> 422
     There is no customer C9999.
3. correct form        -> 302 /Bills/Detail/B2954
   the bill page       -> 200
     Bill B2954 created: 4 x Rp 8.000 = Rp 32.000.
4. Pay      50000 -> 302; status queued, balance Rp 32.000; The amount must be from Rp 1 to the balance, Rp
32.000.
5. Pay      20000 -> 302; status queued, balance Rp 12.000; Payment of Rp 20.000 recorded.
6. Advance        -> 302; status finished, balance Rp 12.000; Bill B2954 is now finished.
7. Advance        -> 302; status departed, balance Rp 12.000; Bill B2954 is now departed.
8. Advance        -> 302; status departed, balance Rp 12.000; Bill B2954 has already been collected.
9. no token (forged)   -> 400
```

The run of `bills_check.py` against the Razor pages; the Java and Python versions print exactly the same lines. Every POST in lines 4 to 8 answered 302, and the forged request 400. Book: Sections 12.6.1 and 12.6.2.

Problem 12.9: Thin Client and Fat Client

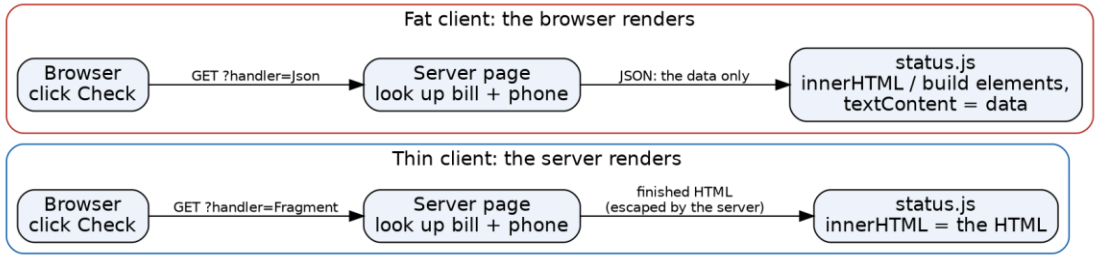


Figure 12.8 Thin client and fat client. The question and the lookup are the same; what differs is whether the server or the browser turns the data into HTML.

***Figure 12.8**: the question and the lookup are the same; what differs is whether the server or the browser turns the data into HTML. The bill is shown only to someone who also knows the phone. Book: Section 12.7.1.*

The Tempting Way: innerHTML

Listing 12.11 The tempting mode of `status.js`: the JSON glued into an HTML string and assigned to `innerHTML`.

```
// Fat client, the tempting way: glue the data into HTML text. A name with markup in it RUNS.
async fatUnsafe() {
  const r = await fetch(query("Json")), d = await r.json();
  answer.innerHTML = r.ok ? `<section class="card"><h2>${d.bill} for ${d.customer}</h2>
    <p>${d.service}. Status: <b>${d.status}</b>. Balance ${d.balance}.</p></section>`
    : `<p class="err">${d.error}</p>`;
},
```

The thin mode pastes the server's fragment, which Razor already escaped: `answer.innerHTML = await (await fetch(query("Fragment"))).text();`.

***Listing 12.11**: a template string looks like a Razor template but escapes nothing; a name with markup in it is parsed and run. Book: Section 12.7.1.*

The Right Way: textContent

Listing 12.12 The safe fat mode of `status.js`: elements built with `createElement`, the data put in with `textContent`.

```
// Fat client, the right way: build elements and set textContent, which is never parsed as HTML.
async fat() {
  const r = await fetch(query("Json")), d = await r.json();
  const box = document.createElement(r.ok ? "section" : "p");
  box.className = r.ok ? "card" : "err";
  if (r.ok) {
    const h2 = document.createElement("h2"), p = document.createElement("p"), b = document.createElement("b");
    h2.textContent = `${d.bill} for ${d.customer}`;
    b.textContent = d.status;
    p.append(`${d.service}. Status: `, b, `. Balance ${d.balance}.`);
    box.append(h2, p);
  } else {
    box.textContent = d.error;
  }
  answer.replaceChildren(box);
},
```

***Listing 12.12**: `textContent` is never parsed as HTML, so no value can become markup. The same `status.js`, byte for byte, serves the Razor, Java and Flask versions. Book: Section 12.7.1.*

What Travels, and What Runs

```
Fragment  165 bytes:   <section class="card">
    <h2>B2950 for Ma&#x27;ruf Saputra</h2>
    <p>Wash, dry &amp; fold (per kg). Status: <b>queued</b>. Balance Rp 48.000.</p>
  </section>

Json      132 bytes: {"bill":"B2950","customer":"Ma\u0027ruf Saputra","service":"Wash, dry \u0026 fold (per
kg)","status":"queued","balance":"Rp 48.000"}
Check (thin)              shows: 'B2952 for <img src=x onerror="document.title=\'hacked\'">'  markup ran: no
Check (fat, innerHTML)    shows: 'B2952 for '                                                 markup ran: YES
Check (fat, textContent)  shows: 'B2952 for <img src=x onerror="document.title=\'hacked\'">'  markup ran: no
```

The run of `status_demo.py`: 165 bytes of HTML against 132 of JSON; for a customer whose "name" is an `<img onerror=…>`, only the `innerHTML` mode ran the markup. Book: Section 12.7.1.

Cross-Site Scripting, Caught in the Act



Figure 12.9 The unsafe fat client after one click: the customer's "name" is now an element of the page, and its script has run.

***Figure 12.9**: the name became an element of the page and its script ran — in a real attack, reading the page or sending requests with the customer's session. The server's JSON was not at fault; one line of script was. Book: Section 12.7.1.*

Problem 12.10: the Same Page on Five Systems

Listing 12.13 The query of `queue.php`: every outlet appears, even with nothing waiting — `LEFT JOIN`, the conditions in `ON`, `COUNT(b.id)`.

```
$service = $_GET['service'] ?? 'S0003';
// Every outlet appears, even with nothing waiting: LEFT JOIN, the conditions in ON, COUNT(b.id).
$st = $db->prepare("SELECT o.id, o.name, COUNT(b.id) AS queued
                      FROM Outlet o
                      LEFT JOIN Bill b ON b.Outlet_id = o.id AND b.status = 'queued' AND b.Service_id = ?
                     GROUP BY o.id, o.name
                     ORDER BY o.id");
$st->execute([$service]);
```

***Listing 12.13**: PHP, JSP, Razor Pages (a LINQ `GroupJoin`), Java and Flask printed the same table: two ironing bills waiting at Rungkut, none at the other two outlets. Book: Section 12.7.2.*

Three Tempting Queries and the Right One

```
A. INNER JOIN                              O01=2
B. LEFT JOIN, conditions in WHERE          O01=2
C. LEFT JOIN, COUNT(*)                     O01=2  O02=1  O03=1
D. LEFT JOIN, conditions in ON, COUNT(b.id)  O01=2  O02=0  O03=0
```

A. An ordinary `JOIN` keeps only outlets with a match: two vanish. **B.** Conditions in `WHERE` run after the join and undo the `LEFT JOIN`.

C. `COUNT(*)` counts the row of nulls: Keputih and Wiyung get 1 instead of 0 — the most dangerous answer, because it looks plausible.

Start from the table whose rows must all appear, put the conditions in `ON`, and count a column of the joined table. Book: Section 12.7.2.

Problem 12.11: a Form That Cannot Be Forged

Razor Pages adds `__RequestVerificationToken` to every POST form and checks it before any handler: 400 otherwise.

The session token (the Back Office, Flask): a random value in the session, printed into every form, compared on every POST.

The double-submit cookie (the Java bill pages): the value in a cookie of its own and in the form; the server keeps nothing and compares the two.

```
POST with no token                  -> 400
POST with the attacker's own token  -> 400
POST with the victim's own token    -> 302 /Bills/Detail/B2955
```

Sent with the victim's cookies: no token and the attacker's own token were refused; only the genuine form passed. Checking only for `POST`, checking `Referer`, one token for everybody, or `[IgnoreAntiforgeryToken]` are all tempting and wrong. Book: Section 12.8.1.

The Problems Compared

Table 12.6 The problems of the chapter compared, first part: the project and the Web Forms page, with the case each tempting solution gets wrong.

Problem	Topic	Main idea	What the tempting solution gets wrong
12.1	data model	print the physical model from the database	a hand-drawn table goes stale; declared keys not enforced
12.2	eight CRUD features in JSP	pages call one model class; guard, PRG, `c:out`	a delete link is a `GET` that anything can follow
12.3	“beautiful”	named colors, measured contrast, one step, one scale	light grey text at 2.50 : 1
12.4	evidence and delivery	acceptance test over HTTP, WAR, container, `.gitignore`	screenshots taken once prove nothing later
12.5	Web Forms to Razor Pages	page model, binding, attributes, server-side price	a total computed by the browser
12.6	application state	one lock for every request	read-add-write races; an empty file read as 0

***Table 12.6**: read the last column — every tempting solution works on the ordinary example and fails in one particular situation. Book: Section 12.9.*

The Problems Compared, Continued

Table 12.7 The problems of the chapter compared, second part: state, forms, handlers, clients, five systems and forged forms.

Problem	Topic	Main idea	What the tempting solution gets wrong
12.7	validated form	shape rules as attributes, existence rules in code	trusting the browser's `required` and `pattern`
12.8	handlers	named handlers, every POST ends in a redirect	answering a POST with a page; Reload repeats it
12.9	thin and fat clients	who makes the HTML; `textContent`	data glued into `innerHTML` runs as code
12.10	five systems	`LEFT JOIN`, conditions in `ON`, `COUNT(b.id)`	`JOIN`, `WHERE` or `COUNT(*)` loses or invents rows
12.11	forged forms	a per-browser token on every POST	one token for all, `Referer` checks, switching it off

***Table 12.7**: two visitors at once, markup in a name, an outlet with nothing waiting, a request the user never started — none is the example in the statement. Write the awkward cases down before the code. Book: Section 12.9.*

Profitina in Practice

A fat client with a duty: one `GET /api/me/dashboard` returns JSON, and every text another user wrote is put on the page as text, never as markup.

Optimistic forms, still checked on the server; links pass a server-side safety screen.

Limits that hold when requests arrive together — channels per day, login attempts — decided in the database, in one transaction.

No release goes out until well over a thousand tests are green and the new version answers its health check. The runnable version of every idea is the Profitina Laundry code. Book: Section 12.10.

Common Mistakes

Design: writing the page first; SQL and rules in JSP scriptlets; data and secrets in the repository.

Requests: changing data with a `GET`; answering a `POST` with a page; weakening the anti-forgery token.

Checks and state: trusting the browser's checks; forgetting `ModelState.IsValid`; read-add-write without a lock; foreign keys not enforced.

Output and SQL: data in `innerHTML`; `JOIN`, `WHERE` or `COUNT(*)` where a `LEFT JOIN` count is meant; low-contrast text; evidence that cannot be repeated.

Each mistake is listed in the book with the problem that shows it. Book: Section 12.11.

Summary: Five Things to Remember

1. Six questions before the page: requests, state, permissions, checks, escaping, proof.

2. One model class under thin pages; the physical model printed by the database; a design whose contrast is computed.

3. An acceptance test over HTTP proves every day what a screenshot proves once.

4. Shared state needs one lock or the database; rules and prices on the server; every POST ends in a redirect.

5. `textContent`, not `innerHTML`; `LEFT JOIN` with `COUNT(b.id)`; one anti-forgery token per browser on every POST.

Book: Section 12.12.

Next Time

Chapter 13 — Data on the Page: GridView and Data Grids. Tables of data, from the Web Forms GridView to its modern form.

Before then: work Exercises 12.1, 12.3, 12.5 and 12.9.

The solutions to all ten exercises are in the book. Book: Sections 12.13 and 12.14.

Lecture 13 — Data on the Page: GridView and Data Grids

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 13

Lecturer: Irfan Subakti — yifana@gmail.com

Today a web page stops showing one record and starts showing the records: 2,953 bills that people page, sort, filter, edit, delete and export. Book: Chapter 13, Sections 13.1–13.25.

What This Meeting Covers

1. **The problem** — the seven jobs of a data grid, and the data underneath: MyLaundry and Profitina Laundry.
2. **GridView** — columns, commands, postbacks, templates, data source controls, and the bugs worth learning from.
3. **The modern grid** — state in the address, bound filters, a white list for sorting, `LIMIT ? OFFSET ?`.
4. **Speed** — why page 119 is slower than page 1, keyset paging, and measuring both.
5. **Changing data** — inline status edits with optimistic concurrency, confirmed deletes, a safe CSV export.
6. **Four tracks and one script** — Razor Pages, PHP, Java, Flask, progressive enhancement with `fetch()`, one set of checks.

The order follows the chapter: the classic control first, because the modern grid is easiest to understand as the answer to its limits. Book: Section 13.1.

Where We Left Off

Chapter 9 introduced ASP.NET: Web Forms (2002, Windows and .NET Framework only) and ASP.NET Core with Razor Pages, which this book runs. Chapter 10 compared PHP, JSP and ASP.NET; Chapter 11 was forms; Chapter 12 worked problems across all three.

A form puts **one** record into the database. The owner of Profitina Laundry wants to see **the bills**: this week's, the ones waiting at Keputih, the most expensive, the ones typed by mistake this morning.

> **DEFINITION.** A **data grid** is an HTML table generated from a data source, one row per record and one column per field, together with the controls that let a person work with it: moving between pages, sorting by a column, filtering rows, editing or deleting a row in place, and exporting what is shown.

Every control of a grid is a link or a small form, and every one ends as an ordinary HTTP request. A grid is where carelessness with a lot of data becomes expensive. Book: Section 13.1.

Seven Jobs

Table 13.1 The seven jobs of a data grid, in Web Forms and in this chapter's grid.

Job	What the person wants	GridView (Web Forms)	This chapter's grid
Show	the records as rows and columns	`DataSource` + `DataBind()`, `BoundField`	an HTML `<table>` drawn by a template
Page	25 rows at a time, and a way to move on	`AllowPaging`, `PageSize`, `PageIndex`	`LIMIT ? OFFSET ?` (and keyset paging, Section 13.13)
Sort	by any column, either direction	`AllowSorting`, `SortExpression`	`?sort=customer&dir=asc`, from a white list
Filter	only some rows: an outlet, a status, some dates	a data source control with `SelectParameters`	`?outlet=O02&status=queued&from=…&to=…`, bound
Edit	change one field of one row, where it stands	`EditIndex`, `EditItemTemplate`, `RowUpdating`	`?edit=B2934`, then `POST /bills/status`
Delete	remove a row, after a question	`CommandField`, `RowDeleting`, a `confirm()`	`?delete=B2931` or a `confirm()`, then `POST /bills/delete`
Export	take the rows away in a file	not built in	`GET /bills.csv?…`, the same filters

The new difficulty in Table 13.1 is doing all seven on one page: a sort keeps the filter, a page link keeps both. Book: Section 13.2.1 and Table 13.1.

The Size of the Problem

Table 13.2 The 2,953 bills of the sample database by outlet and status, counted with `GROUP BY`.

	Rungkut (O01)	Keputih (O02)	Wiyung (O03)	All outlets
queued	9	9	5	23
finished	23	6	5	34
departed	1,280	981	635	2,896
all statuses	**1,312**	**996**	**645**	**2,953**

As one HTML table the grid would be about 1.2 MB (roughly 400 bytes a row); 25 at a time it is 119 pages of about 14 KB.

In Table 13.2 only 57 bills are still in the shop, which is why filters are not a luxury. MyLaundry has four bills; on four rows paging and sorting are decoration. Book: Section 13.2.2 and Table 13.2.

The MyLaundry Design

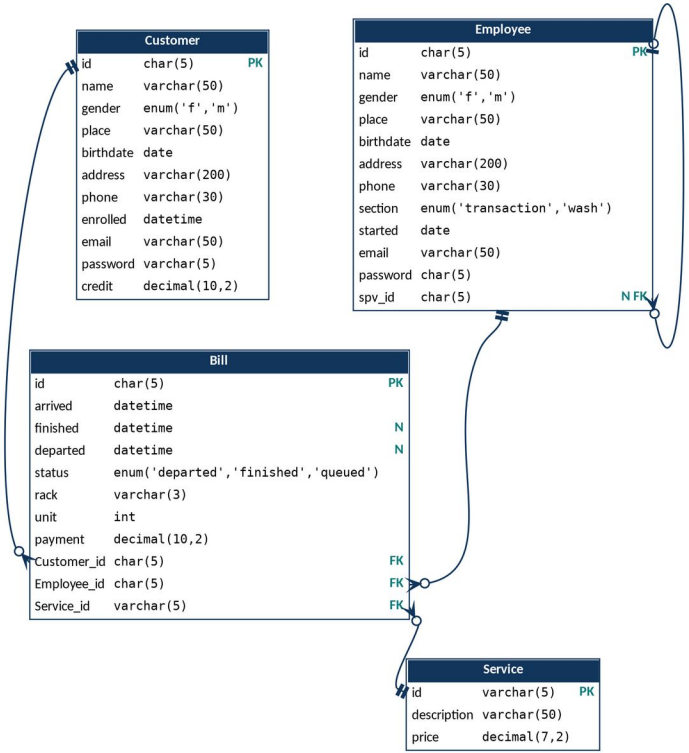


Figure 13.1 The MyLaundry design. A Bill belongs to one Customer, was received by one Employee and is for one Service; an Employee may have a supervisor who is another Employee (the loop on the right). `N` marks a column that may be NULL.

The case study: "Implement a web program for MyLaundry … The most noticeable feature for the customer is the list of services and their prices." In Figure 13.1 read the crow's feet as "many"; `finished` and `departed` may be NULL while the laundry is still in the shop. Book: Sections 13.2.3 and 13.3.1.

The Bill Table

Listing 13.1 The database and its Bill table, from `MyLaundryDB.sql`. `status` is an `enum` of three words, and a new bill is `queued` unless it says otherwise.

```
CREATE DATABASE MyLaundryDB;
USE MyLaundryDB;

-- Table: Bill
CREATE TABLE Bill (
    id char(5) NOT NULL,
    arrived datetime NOT NULL DEFAULT CURRENT_TIMESTAMP(),
    finished datetime NULL,
    departed datetime NULL,
    status enum('departed','finished','queued') NOT NULL DEFAULT 'queued',
    rack varchar(3) NOT NULL,
    unit int NOT NULL,
    payment decimal(10,2) NOT NULL,
    Customer_id char(5) NOT NULL,
    Employee_id char(5) NOT NULL,
    Service_id varchar(5) NOT NULL,
    CONSTRAINT Bill_pk PRIMARY KEY (id)
);
```

In Listing 13.1, MySQL sorts an `enum` by the position of its values; listed alphabetically, they sort exactly as SQLite sorts the same words stored as text, which the grid relies on later. phpMyAdmin, which shows the Service table with Edit, Copy and Delete on every row, is itself a data grid. A console program reads the same table with a data reader. Book: Sections 13.3.1 and 13.3.2, Figures 13.3 and 13.4.

Profitina Laundry's Bills

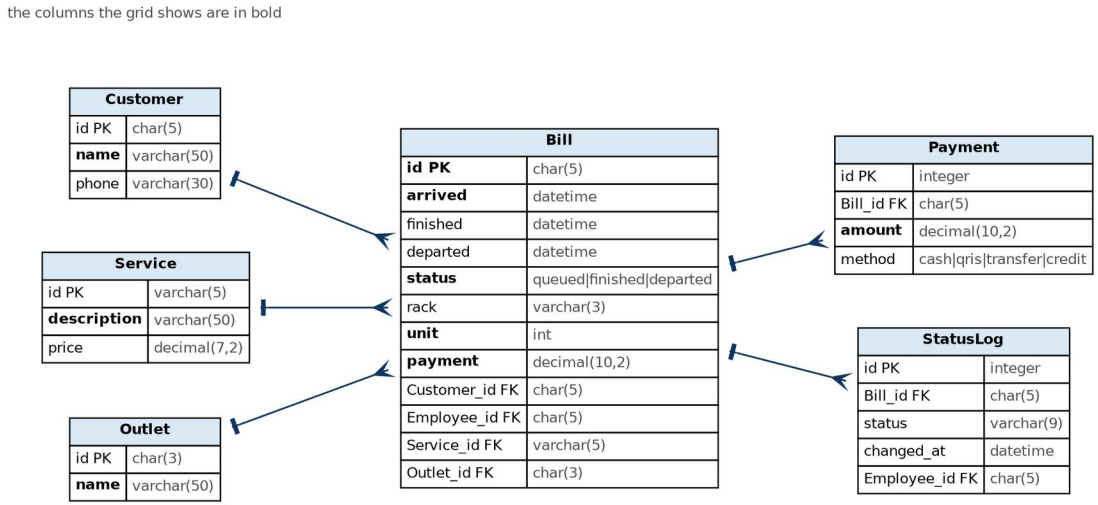


Figure 13.2 The part of the Profitina Laundry database behind the bills grid. The grid joins Bill to Customer, Service and Outlet for its columns, sums Payment for the *Paid* column, and writes StatusLog whenever it changes a status.

Three schema rules in Figure 13.2 bind the grid: a CHECK constraint ties the status to the `finished`/`departed` timestamps; every foreign key into Bill is `ON DELETE RESTRICT`; and `payment` is the amount charged, while paid is the sum of Payment rows. Book: Section 13.3.3.

The Smallest GridView

A **GridView** renders a data source as an HTML `<table>`; nothing appears until it is **bound** with `DataBind()`. Listing 13.2 is the whole page.

Listing 13.2 `01Show.aspx`: a form with one GridView and no columns declared. Not run in this book's build (Web Forms needs Windows and .NET Framework 4.8).

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="01Show.aspx.cs" Inherits="_01Show" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Show the grid</title>
</head>
<body>
    <form id="form1" OnInit="Page_Load" runat="server">
        <asp:GridView ID="GridView1" runat="server">
        </asp:GridView>
    </form>
</body>
</html>
```

A server control is turned into an object on the server and asked to render itself as HTML at the end. `AutoGenerateColumns` *is on by default: one column per data column, headed with its name. Book: Sections 13.4.1 and 13.4.2.*

Filling and Binding the Grid

Listing 13.3 `Page_Load` in `01Show.aspx.cs`: fill a DataSet from the Service table, bind the grid to it, and style the header and alternating rows. Not run in this book's build.

```
    protected void Page_Load(object sender, EventArgs e) {
        string cs = "server=localhost;userid=root;password=;database=MyLaundryDB";
        MySqlConnection con = new MySqlConnection(cs);
        cs = "SELECT * FROM Service";
        MySqlDataAdapter adapter = new MySqlDataAdapter(cs, con);
        con.Close();
        DataSet ds = new DataSet();
        adapter.Fill(ds, "Service");
        if (ds.HasErrors) {
            ds.RejectChanges();
        } else {
            ds.AcceptChanges();
        }
        GridView1.DataSource = ds.Tables["Service"].DefaultView;
        GridView1.DataBind();
```

In Listing 13.3 the data adapter's `Fill` opens and closes the connection itself, so the early `con.Close()` does nothing. `DefaultView` is a DataView that can sort and filter in memory. What reaches the browser is an ordinary `<table>`. Book: Section 13.4.2.

The Price List, Drawn

id	description	price
B120	Bed cover, size 120	30000.00
B180	Bed cover, size 160-180	40000.00
C	Carpet, per m2	10000.00
C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00
C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00
C10W	Clothes 7-10 kg, Washed only	25000.00
C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00
C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00
C4W	Clothes max 4 kg, Washed only	10000.00
C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00
C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00
C6W	Clothes 5-6 kg, Washed only	15000.00
DL1	Doll, Large 1 pcs	40000.00
DM2	Doll, Medium 2 pcs	40000.00
DO	Delivery Order, per 10 kg	5000.00
DS3	Doll, Small 3 pcs	40000.00
I	Ironed only (without washing), per 5 kg	10000.00

Figure 13.3 The MyLaundry price list drawn by a GridView: a header row styled light blue and alternating rows styled light yellow, set from three lines of C# in Listing 13.3.

Figure 13.3 is small and plain, and it is still what most of the business web looks like on the inside: almost every administrative page is this picture with more columns. Book: Sections 13.2.3 and 13.4.2.

Declaring the Columns

Table 13.3 The GridView's column types. `BoundField` and `TemplateField` do nearly all the work; the others are conveniences.

Field	Draws in each row	Typical attributes
`BoundField`	the value of one data column, as text	`DataField`, `HeaderText`, `DataFormatString`, `ReadOnly`, `SortExpression`
`ButtonField`	a button or link that raises a command	`ButtonType`, `Text`, `CommandName`
`CommandField`	the standard Select, Edit, Update, Cancel and Delete links	`ShowEditButton`, `ShowDeleteButton`, `ShowSelectButton`
`TemplateField`	whatever markup you write, per mode	`ItemTemplate`, `EditItemTemplate`, `HeaderTemplate`, `FooterTemplate`
`CheckBoxField`	a check box for a true/false column	`DataField`
`HyperLinkField`	a link built from data	`DataNavigateUrlFields`, `DataNavigateUrlFormatString`
`ImageField`	an image whose address is data	`DataImageUrlField`

The second page, `02DeleteSelect.aspx`, turns `AutoGenerateColumns` off and declares three `BoundField`s and two `ButtonField`s from Table 13.3. The command names `Delete` and `Select` become the events `RowDeleting` and `SelectedIndexChanged`. Book: Section 13.5.1.

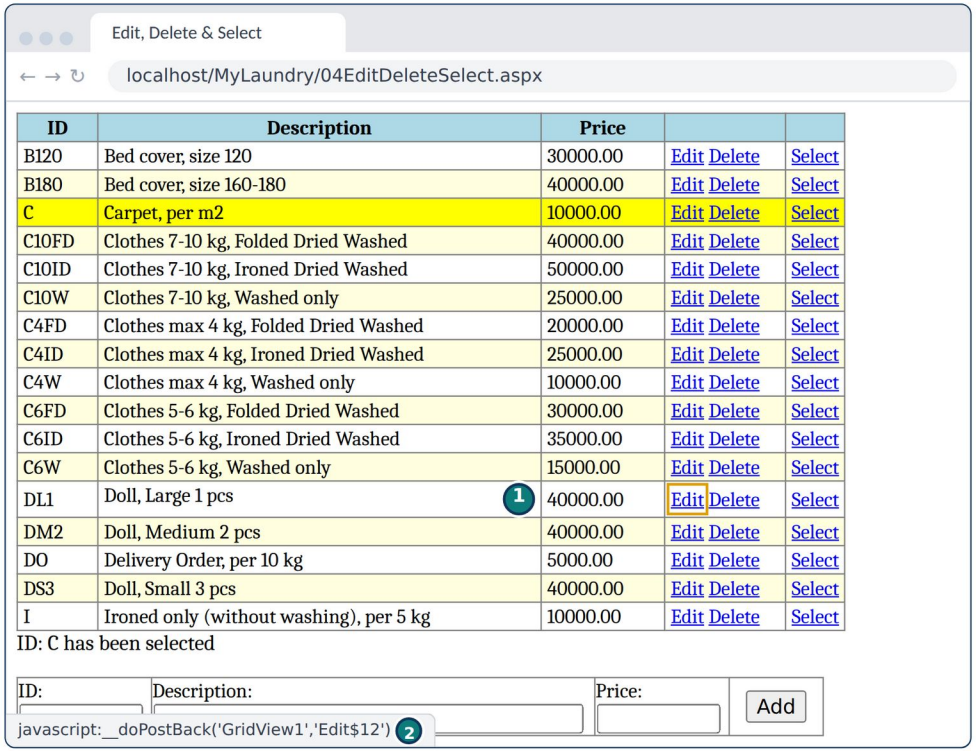
The `DeleteRow` Handler

Listing 13.4 The start of `DeleteRow` in `02DeleteSelect.aspx.cs`: remove the row from the view, then from the database. Not run in this book's build.

```
    protected void DeleteRow(object sender, GridViewDeleteEventArgs e) {
        // Save the "id" which will be deleted later
        string deletedID = GridView1.Rows[e.RowIndex].Cells[0].Text;
        dataView.Delete(e.RowIndex);
        GridView1.DataSource = dataView;
        GridView1.DataBind();
        // It's only delete a row in DataView not on the table
        // So, if you also want to delete it on the DB's table, it needs more processes
        string cs = "server=localhost;userid=root;password=;database=MyLaundryDB";
        MySqlConnection con = new MySqlConnection(cs);
        try {
            con.Open();
            cs = "DELETE FROM Service WHERE id='" + deletedID + "'";
            MySqlCommand cmd = new MySqlCommand(cs, con);
            cmd.ExecuteNonQuery();
```

Read Listing 13.4 slowly: the id comes from what the grid displayed in the first cell, and the `DELETE` is built by joining strings. Both are mistakes, and Worked Example 1 shows why. Book: Sections 13.5.1 and 13.5.3.

A Link That Calls C#



ID	Description	Price		
B120	Bed cover, size 120	30000.00	Edit Delete	Select
B180	Bed cover, size 160-180	40000.00	Edit Delete	Select
C	Carpet, per m2	10000.00	Edit Delete	Select
C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00	Edit Delete	Select
C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00	Edit Delete	Select
C10W	Clothes 7-10 kg, Washed only	25000.00	Edit Delete	Select
C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00	Edit Delete	Select
C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00	Edit Delete	Select
C4W	Clothes max 4 kg, Washed only	10000.00	Edit Delete	Select
C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00	Edit Delete	Select
C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00	Edit Delete	Select
C6W	Clothes 5-6 kg, Washed only	15000.00	Edit Delete	Select
DL1	Doll, Large 1 pcs	40000.00	Edit Delete	Select
DM2	Doll, Medium 2 pcs	40000.00	Edit Delete	Select
DO	Delivery Order, per 10 kg	5000.00	Edit Delete	Select
DS3	Doll, Small 3 pcs	40000.00	Edit Delete	Select
I	Ironed only (without washing), per 5 kg	10000.00	Edit Delete	Select

ID: C has been selected

ID:	Description:	Price:	Add

Figure 13.4 The pointer on the *Edit* link of row DL1 (marker 1). The status bar (marker 2) shows where the link goes: `javascript:__doPostBack('GridView1','Edit$12')`, a script call rather than an address.

In Figure 13.4 the link goes to no address: `__doPostBack` writes the control and the argument (`Edit$12`, the Edit command on row 12, counting from 0) into two hidden fields, `__EVENTTARGET` and `__EVENTARGUMENT`, and submits the page's one form. Book: Section 13.5.2.

One Click, One Postback

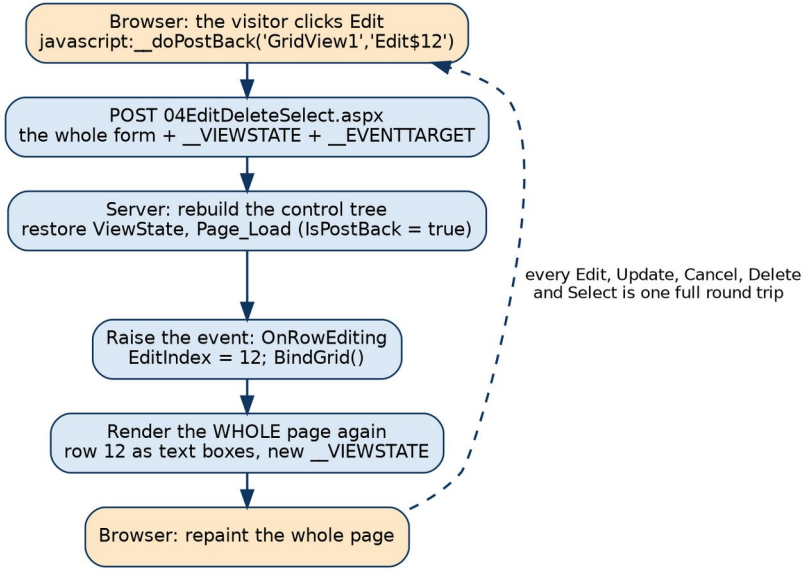


Figure 13.5 One click on a GridView, as Web Forms handles it. Every Select, Edit, Update, Cancel and Delete is a full POST of the whole form with its view state, and a full new page in reply.

Figure 13.5: ASP.NET rebuilds the control tree, restores state from `__VIEWSTATE`, runs `Page_Load` with `IsPostBack` true, raises the event, and renders the entire page again. Book: Section 13.5.2.

What a Postback Costs

Weight. A grid's view state carries enough to redraw every row, and every click posts it back.

No address. Every action is a POST to the same URL: a filtered, sorted, paged grid cannot be bookmarked or shared, and Back asks to resubmit.

Session copies. `02DeleteSelect` keeps the DataSet in the session: one query saved per click, memory spent per visitor, stale as soon as anyone else changes the table.

> **THE KEY INSIGHT.** Web Forms did not avoid HTTP; it hid it. Every GridView event is an HTTP POST, and every modern grid in this chapter makes the same requests visible: a link for each way of *looking*, a POST only for each way of *changing*.

The design was brilliant in 2002, when it let desktop programmers write event handlers instead of learning HTTP. Book: Sections 13.5.2 and 13.5.3.

Worked Example 1: Data Becomes Code

Step 1. `DeleteRow` builds its statement by joining strings:

```
cs = "DELETE FROM Service WHERE id='" + deletedID + "'";
```

Step 2. Someone uses the *Add* form of the fourth page, which *does* use parameters, to add a service whose id is the five characters `'OR'1`. It is stored faithfully.

Step 3. A week later a colleague deletes that service on this page. The statement becomes `DELETE FROM Service WHERE id=''OR'1'`: "id is empty, or true". Every row matches.

*Data that was stored safely has become code on the way out: **second-order SQL injection**. The only reliable defense is the same as for the first-order kind: never splice a value into SQL text; bind it. Book: Section 13.5.3.*

Worked Example 1: Saved by Accidents

Accident 1. A `BoundField` HTML-encodes what it shows, so the cell holds `'OR'1` and the spliced statement matches nothing. Turn off `HtmlEncode` (a common "fix" for `&`) and it matches everything.

Accident 2. Four services are used by bills, so the foreign keys make MySQL refuse the whole statement. On a table nothing refers to, every row goes.

The second fault. `Cells[0].Text` is *rendered*, encoded text: an honest id with `&` or `'` also reaches the SQL changed. The key belongs in the grid's data keys, not its display.

> **Your own database is not a trusted source.** A value that was stored safely with parameters is still just text. Joining it into a later SQL statement turns it into code. Bind every value, whether it came from a form a minute ago or from a table last year.

Code whose safety rests on accidents is not safe. Exercise 13.1 writes the two corrected lines. Book: Sections 13.5.3 and 13.27.

Templates and Data Keys

Listing 13.5 From the grid of `04EditDeleteSelect.aspx`: `DataKeyNames`, six event handlers, and the Description column with an item template and an edit template. Not run in this book's build.

```
        <asp:GridView ID="GridView1" runat="server"
            AutoGenerateColumns="false"
            DataKeyNames="id" OnSelectedIndexChanged = "SelectRow"
            OnRowDataBound="OnRowDataBound" OnRowEditing="OnRowEditing"
            OnRowCancelingEdit="OnRowCancelingEdit" OnRowUpdating="OnRowUpdating"
            OnRowDeleting="OnRowDeleting" EmptyDataText="No records has been added." style="margin-top:
0px">
            <Columns>
                <asp:TemplateField HeaderText="Description" ItemStyle-Width="400">
                    <ItemTemplate>
                        <asp:Label ID="lblDescription" runat="server" Text='<%#
DataBinder.Eval(Container.DataItem, "Description") %>'></asp:Label>
                    </ItemTemplate>
                    <EditItemTemplate>
                        <asp:TextBox ID="txtDescription" runat="server" Text='<%#
DataBinder.Eval(Container.DataItem, "Description") %>'></asp:TextBox>
                    </EditItemTemplate>
                </asp:TemplateField>
```

In Listing 13.5, `DataKeyNames="id"` keeps each row's key in view state, so the key comes from the data, not the display. `<%# … %>` is a data-binding expression; Description is a `Label` normally and a `TextBox` in edit mode, while ID has no edit template. Book: Section 13.6.1.

The Events That Edit

Table 13.4 The GridView events that matter for editing. Each is one postback (Figure 13.5); the "-ing" events can be cancelled by setting `e.Cancel = true`.

Event	Raised when	This page uses it to
`RowDataBound`	each row has just been built from its data	attach a `confirm()` to the Delete link
`RowEditing`	an Edit link was clicked	set `EditIndex` and rebind
`RowCancelingEdit`	a Cancel link was clicked	set `EditIndex = -1` and rebind
`RowUpdating`	an Update link was clicked	read the text boxes, `UPDATE`, leave edit mode
`RowDeleting`	a Delete link was clicked (and the dialog said OK)	`DELETE` by the row's key, rebind
`SelectedIndexChanged`	a Select link was clicked	color the row, show its key
`PageIndexChanging`, `Sorting`	a pager link or a heading was clicked	(not used here; Section 13.7)

Editing is one number, the grid's `EditIndex`: the row being edited, or -1 for none. `OnRowEditing` sets it and rebinds; `OnRowCancelingEdit` sets it back to -1. Every handler in Table 13.4 ends by rebinding the grid. Book: Sections 13.6.1 and 13.6.2.

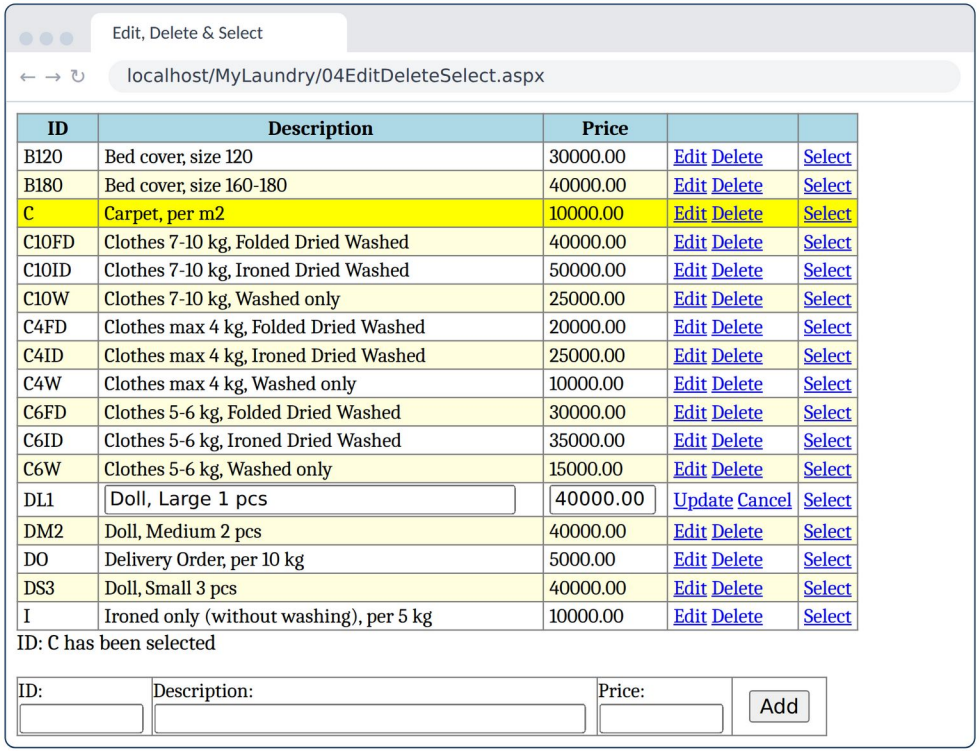
Updating a Row

Listing 13.6 The start of `OnRowUpdating` in `04EditDeleteSelect.aspx.cs`: find the text boxes, take the key from `DataKeys`, and run a parameterized `UPDATE`. Not run in this book's build.

```
    protected void OnRowUpdating(object sender, GridViewUpdateEventArgs e) {
        GridViewRow row = GridView1.Rows[e.RowIndex];
        string id = GridView1.DataKeys[e.RowIndex].Values[0].ToString();
        string description = (row.FindControl("txtDescription") as TextBox).Text;
        string price = (row.FindControl("txtPrice") as TextBox).Text;
        using (MySqlConnection con = new MySqlConnection(cs)) {
            using (MySqlCommand cmd = new MySqlCommand("UPDATE Service SET Description "
                + "= @Description, Price = @Price WHERE ID = @ID")) {
                using (MySqlDataAdapter sda = new MySqlDataAdapter()) {
                    cmd.Parameters.AddWithValue("@ID", id);
                    cmd.Parameters.AddWithValue("@Description", description);
                    cmd.Parameters.AddWithValue("@Price", price);
```

Listing 13.6 binds every value; the `using` blocks close connection, command and adapter even after an exception. After the update, phpMyAdmin shows DL1 as Doll, Large 1 pcs (special). Book: Section 13.6.2 and Figure 13.9.

Row DL1 in Edit Mode



ID	Description	Price		
B120	Bed cover, size 120	30000.00	Edit Delete	Select
B180	Bed cover, size 160-180	40000.00	Edit Delete	Select
C	Carpet, per m2	10000.00	Edit Delete	Select
C10FD	Clothes 7-10 kg, Folded Dried Washed	40000.00	Edit Delete	Select
C10ID	Clothes 7-10 kg, Ironed Dried Washed	50000.00	Edit Delete	Select
C10W	Clothes 7-10 kg, Washed only	25000.00	Edit Delete	Select
C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00	Edit Delete	Select
C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00	Edit Delete	Select
C4W	Clothes max 4 kg, Washed only	10000.00	Edit Delete	Select
C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00	Edit Delete	Select
C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00	Edit Delete	Select
C6W	Clothes 5-6 kg, Washed only	15000.00	Edit Delete	Select
DL1	Doll, Large 1 pcs	40000.00	Update Cancel	Select
DM2	Doll, Medium 2 pcs	40000.00	Edit Delete	Select
DO	Delivery Order, per 10 kg	5000.00	Edit Delete	Select
DS3	Doll, Small 3 pcs	40000.00	Edit Delete	Select
I	Ironed only (without washing), per 5 kg	10000.00	Edit Delete	Select

ID: C has been selected

ID:	Description:	Price:	
			Add

Figure 13.6 Row DL1 in edit mode: Update and Cancel have replaced Edit and Delete, and row C is still selected from an earlier click. The ID has no edit template in Listing 13.5, so it stays plain text.

Figure 13.6: Update and Cancel have replaced Edit and Delete in the row whose index is `EditIndex`. Book: Section 13.6.2.

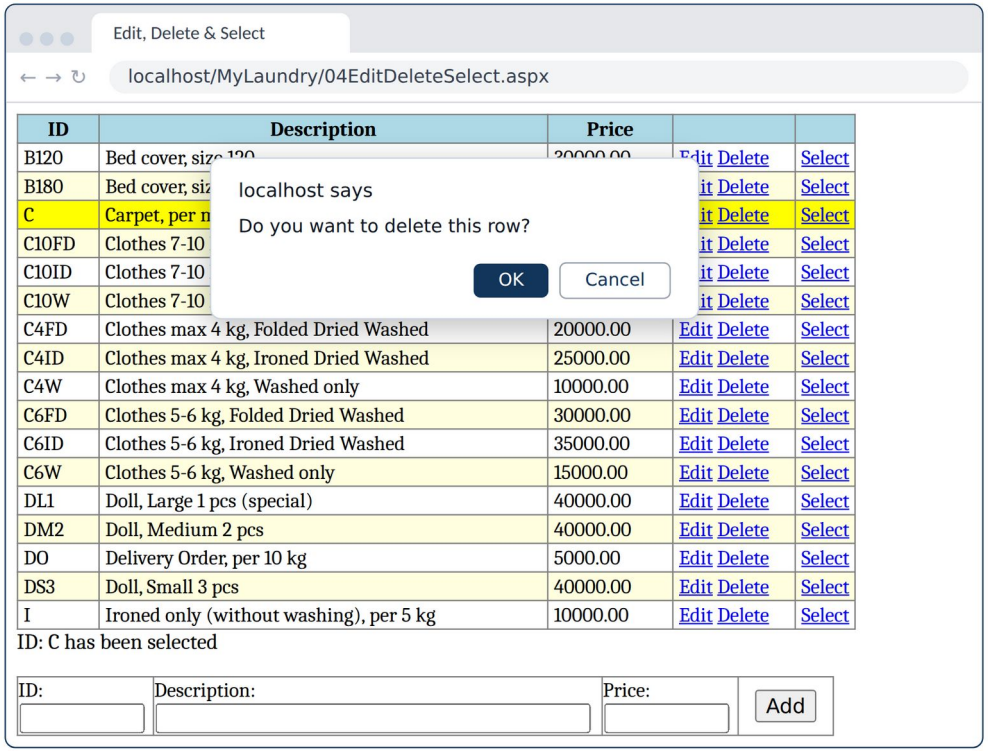
Deleting, After a Question

Listing 13.7 `OnRowDataBound` and the start of `OnRowDeleting` in `04EditDeleteSelect.aspx.cs`: a `confirm()` on every Delete link, and a delete by the row's key with a parameter. Not run in this book's build.

```
    protected void OnRowDataBound(object sender, GridViewRowEventArgs e) {
        if (e.Row.RowType == DataControlRowType.DataRow && e.Row.RowIndex != GridView1.EditIndex) {
            (e.Row.Cells[3].Controls[2] as LinkButton).Attributes["onClick"] =
                "return confirm('Do you want to delete this row?');";
        }
    }
    protected void OnRowDeleting(object sender, GridViewDeleteEventArgs e) {
        string id = (string) GridView1.DataKeys[e.RowIndex].Values[0];
        using (MySqlConnection con = new MySqlConnection(cs)) {
            using (MySqlCommand cmd = new MySqlCommand("DELETE FROM Service WHERE ID = @ID")) {
                using (MySqlDataAdapter sda = new MySqlDataAdapter()) {
                    cmd.Parameters.AddWithValue("@ID", id);
```

In Listing 13.7, `Controls[2]` is the Delete link, the third control in the command field's cell. The dialog is a convenience for honest people, not a security control: anyone can send the postback without it. Book: Section 13.6.2.

The Browser Asks



ID	Description	Price		
B120	Bed cover, siz… 120	30000.00	Edit Delete	Select
B180	Bed cover, siz…		it Delete	Select
C	Carpet, per n…		it Delete	Select
C10FD	Clothes 7-10		it Delete	Select
C10ID	Clothes 7-10		it Delete	Select
C10W	Clothes 7-10		it Delete	Select
C4FD	Clothes max 4 kg, Folded Dried Washed	20000.00	Edit Delete	Select
C4ID	Clothes max 4 kg, Ironed Dried Washed	25000.00	Edit Delete	Select
C4W	Clothes max 4 kg, Washed only	10000.00	Edit Delete	Select
C6FD	Clothes 5-6 kg, Folded Dried Washed	30000.00	Edit Delete	Select
C6ID	Clothes 5-6 kg, Ironed Dried Washed	35000.00	Edit Delete	Select
C6W	Clothes 5-6 kg, Washed only	15000.00	Edit Delete	Select
DL1	Doll, Large 1 pcs (special)	40000.00	Edit Delete	Select
DM2	Doll, Medium 2 pcs	40000.00	Edit Delete	Select
DO	Delivery Order, per 10 kg	5000.00	Edit Delete	Select
DS3	Doll, Small 3 pcs	40000.00	Edit Delete	Select
I	Ironed only (without washing), per 5 kg	10000.00	Edit Delete	Select

ID: C has been selected

ID:	Description:	Price:	
			Add

Figure 13.7 The browser's confirmation dialog, attached to the Delete link by `OnRowDataBound`. It is drawn by the browser, not by the page, which is why it looks different in every browser.

Figure 13.7: Cancel stops the postback; OK deletes. The Add form puts DL1 back; an empty ID writes an `alert` with `Response.Write`, which breaks the HTML and invites XSS. Book: Sections 13.6.2–13.6.3, Figures 13.11–13.13.

Declaring the Data: `SqlDataSource`

Listing 13.8 The commands of the `dsBills` data source in `LaundryBills/Bills.aspx`: optional filters, and one `UPDATE` that sets the status and its timestamp together. Not run in this book's build.

```
        SelectCommand="SELECT b.id, b.arrived, c.name AS customer, s.description AS service,
                              o.name AS outlet, b.unit, b.payment, b.status
                       FROM Bill b
                       JOIN Customer c ON c.id = b.Customer_id
                       JOIN Service s ON s.id = b.Service_id
                       LEFT JOIN Outlet o ON o.id = b.Outlet_id
                       WHERE (@outlet = '' OR b.Outlet_id = @outlet)
                         AND (@status = '' OR b.status = @status)
                       ORDER BY b.arrived DESC, b.id DESC"
        UpdateCommand="UPDATE Bill SET status = @status,
                              finished = IF(@status = 'finished', NOW(), finished),
                              departed = IF(@status = 'departed', NOW(), departed)
                       WHERE id = @id">
```

In Listing 13.8, `(@outlet = '' OR b.Outlet_id = @outlet)` is true for every row under All outlets. The code behind refuses a skipped step and deletes only queued, unpaid bills. Limits: all 2,953 rows fetched to show 25, no StatusLog row, state hidden in view state. Book: Section 13.7.

From GridView to Razor Pages

Table 13.5 From GridView to a Razor Pages grid: the first six of the eleven rows.

Web Forms GridView	Razor Pages (this chapter)	The web idea underneath
`DataSource` + `DataBind()`	the page model fills a property; the template loops over it with `@foreach`	a server draws HTML from rows
view state (`__VIEWSTATE`)	the query string: `?outlet=002&sort=customer&page=2`	the address *is* the state of a read
a postback for paging and sorting	a link (`GET`)	reading is safe and repeatable
a postback for Update and Delete	a form (`POST`), then a **303** redirect back to the grid	Post/Redirect/Get (Chapter 7)
`SortExpression`	a sort key from a white list, mapped to SQL on the server	never let a request write SQL
`SqlDataSource` paging (in memory)	`LIMIT ? OFFSET ?` in the query	let the database skip rows

ASP.NET Core has no GridView: it would have nothing to stand on. Read the right-hand column of Table 13.5 as the design of the modern grid. Book: Section 13.8.1 and Table 13.6.

Two Kinds of Request

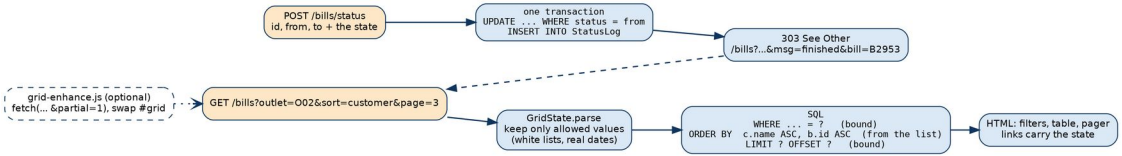


Figure 13.8 The two kinds of request of the bills grid. Reads are GETs whose whole state is in the address; changes are POSTs that end in a 303 redirect back to a read. The optional script (left) makes the same GETs with `fetch()`.

Figure 13.8: a read is a GET whose whole state is in the query string; a change is a POST carrying the bill, its old and new status and the grid's state, answered by a 303 redirect back to a read. Book: Section 13.8.2.

One Contract for Four Grids

Table 13.6 The grid's address scheme: the seven parameters of the grid's state.

Parameter	Allowed values	Default (left out of the address)	Anything else
`outlet`	an id from the Outlet table: `001`, `002`, `003`	all outlets	ignored
`status`	`queued`, `finished`, `departed`	any status	ignored
`from`, `to`	a real date `YYYY-MM-DD`; `to` is inclusive	no limit	ignored
`sort`	`id`, `arrived`, `customer`, `service`, `outlet`, `unit`, `payment`, `status`	`arrived`	ignored
`dir`	`asc`, `desc`	`desc`	ignored
`page`	a whole number from 1	1	below 1 becomes 1; above the last page becomes the last
`size`	10, 25, 50, 100	25	ignored

All four grids answer these addresses (Table 13.6); `edit`, `delete`, `msg`, `bill` and `partial` complete the list. A bad value is simply not used: `?sort=b.id;DROP TABLE Bill` shows the default sort. Book: Section 13.8.3 and Table 13.7.

Parsing the State

Listing 13.9 `GridState.Parse` in `Data/GridState.cs`: every property of the new state has passed a test, or has its default.

```
public static GridState Parse(Func<string, string> get, IReadOnlySet<string> outletIds)
{
    string outlet = get("outlet"), status = get("status"), sort = get("sort"), dir = get("dir");
    int size = int.TryParse(get("size"), out int n) ? n : 25;
    int page = int.TryParse(get("page"), out int p) ? p : 1;
    return new GridState(
        outletIds.Contains(outlet) ? outlet : "",
        Statuses.Contains(status) ? status : "",
        CleanDate(get("from")),
        CleanDate(get("to")),
        SortKeys.Contains(sort) ? sort : "arrived",
        dir is "asc" or "desc" ? dir : "desc",
        Math.Max(1, page),
        Sizes.Contains(size) ? size : 25);
}
```

`GridState` is an immutable C# record (Listing 13.9); `int.TryParse` fails for `"abc"`, `""` and `"99999999999"`. The function argument lets one method read `Request.Query` for a GET and `Request.Form` for a POST. Book: Section 13.9.1.

Building Every Link

Listing 13.10 `Href` and `SortHref` in `Data/GridState.cs`: an address for the state plus extra pairs, and the sort rule every spreadsheet user expects.

```
    /// <summary>An address for this state plus extra pairs such as edit=B0012.</summary>
    public string Href(string path = "/bills", params (string Key, string Value)[] extra)
    {
        var pairs = Params();
        pairs.AddRange(extra.Select(e => new KeyValuePair<string, string>(e.Key, e.Value)));
        return pairs.Count == 0 ? path : path + "?" + string.Join("&",
            pairs.Select(kv => Uri.EscapeDataString(kv.Key) + "=" + Uri.EscapeDataString(kv.Value)));
    }

    /// <summary>Clicking the current column flips the direction; another column starts
ascending.</summary>
    public string SortHref(string key) => key == Sort
        ? (this with { Dir = Dir == "desc" ? "asc" : "desc", Page = 1 }).Href()
        : (this with { Sort = key, Dir = "asc", Page = 1 }).Href();
```

`Params()` *leaves out every default, so one state has one address. In Listing 13.10* `Uri.EscapeDataString` *encodes each value. A POST form carries the state in hidden fields, and the redirect is rebuilt with the same* `Parse`*: a posted* `back=` *address would be an open redirect. Book: Sections 13.9.1 and 13.9.2.*

Filters as Bound Values

Listing 13.11 `where()` in `python/grid.py`: one condition per filter that is set, each with a `?` for its value, and the values in a list in the same order.

```
def where(state):
    """The WHERE clause and its bound values. Only b.* columns are filtered."""
    sql, values = ["1 = 1"], []
    if state.outlet:
        sql.append("b.Outlet_id = ?")
        values.append(state.outlet)
    if state.status:
        sql.append("b.status = ?")
        values.append(state.status)
    if state.date_from:
        sql.append("b.arrived >= ?")
        values.append(state.date_from)
    if state.date_to:                       # 'to' is inclusive: before the next midnight
        sql.append("b.arrived < ?")
        values.append((date.fromisoformat(state.date_to) + timedelta(days=1)).isoformat())
    return " AND ".join(sql), values
```

For `outlet=O02, status=queued`, Listing 13.11 returns `1 = 1 AND b.Outlet_id = ? AND b.status = ?` and `['O02', 'queued']`. Binding makes a value harmless; the white list makes it meaningful. Book: Section 13.10.1.

Worked Example 2: “To” Means the Whole Day

Step 1. `arrived` holds a date and a time, `'2026-09-30 16:26:26'`. The naive `b.arrived <= '2026-09-30'` compares text, and the longer string sorts *after* the shorter one: every bill of the 30th is lost.

Step 2. Compare with the start of the *next* day instead, as Equation (13.1) says:

$$\text{from} \le \text{arrived} < \text{to} + 1\,\text{day} \tag{13.1}$$

Step 3. For September 2026: `'2026-09-01' <= arrived < '2026-10-01'` selects 256 bills; the naive form selects 246, and the ten it drops all have work still to do.

The program adds the day (`date.fromisoformat(...) + timedelta(days=1)`), so the SQL is the same in SQLite and MySQL; `2026-02-30` has the right shape and is dropped. Book: Section 13.10.2.

The Column Cannot Be Bound

Listing 13.12 The output of `bench/order_by_param.py`. Bound, `'c.name'` is one string for every row; written into the statement, `c.name` sorts by name.

```
ORDER BY ?  with 'c.name' bound: [('B0001', 'Imam Pratama'), ('B0002', 'Hendro Ramadhan'), ('B0003',
"Ma'ruf Saputra")]
ORDER BY c.name, b.id written  : [('B0118', 'Adi Eko Alatas'), ('B0119', 'Adi Eko Alatas'), ('B0122',
'Adi Eko Alatas')]
```

Listing 13.13 `SORTS` in `python/grid.py`: eight keys a visitor may name, each mapped to the SQL the program writes.

```python
# The ONLY expressions that may ever follow ORDER BY. A visitor chooses a key;
# the SQL comes from this table, never from the request.
SORTS = {
    "id": "b.id",
    "arrived": "b.arrived",
    "customer": "c.name",
    "service": "s.description",
    "outlet": "o.name",
    "unit": "b.unit",
    "payment": "b.payment",
    "status": "b.status",
}
```

Listing 13.12: placeholders stand for values only. `"ORDER BY " + request.args["sort"]` is one of the most common injection holes. With Listing 13.13 the visitor chooses a key and the program chooses the SQL. Book: Sections 13.11.1 and 13.11.2.

A White List and a Tie-Breaker

Listing 13.14 `order_by()`: the column comes from `SORTS`, the direction is one of two fixed words, and the bill id is always added as a tie-breaker.

```
def order_by(state):
    column = SORTS[state.sort]                      # a KeyError is impossible: parse() checked
    direction = "ASC" if state.dir == "asc" else "DESC"
    if column == "b.id":
        return f"b.id {direction}"
    return f"{column} {direction}, b.id {direction}"   # the tie-breaker makes pages stable
```

Listing 13.15 The real output of `bench/tie_breaker.py`: ten bills sorted by amount, without and with the id as tie-breaker. The last four differ.

```
ORDER BY b.payment DESC             B1222 B1334 B0394 B1361 B1745 B2468 B0044 B0168 B0560 B0615
ORDER BY b.payment DESC, b.id DESC  B1334 B1222 B2468 B1745 B1361 B0394 B2854 B2387 B2377 B2371
bills charged the top amount, Rp 165.000: 2
```

In Listing 13.15 four of seventeen bills tie at Rp 135.000, and SQL promises no order among ties: a bill can appear on two pages while another never appears. Listing 13.14 makes the order total. The rule: a paged query must end its ORDER BY with a unique column. Book: Sections 13.11.2 and 13.11.3.

Paging Is Arithmetic

N matching rows, page size s, page p counted from 1. The number of pages is Equation (13.2):

$$P = \max\left(1, \lceil \frac{N}{s} \rceil\right) \tag{13.2}$$

The offset, the rows skipped before page p, is Equation (13.3):

$$o = (p - 1)\,s \tag{13.3}$$

The rows shown, printed as “Showing 26–50 of 2,953 bills”, are Equation (13.4):

$$\text{first} = o + 1, \text{last} = \min(p\,s, N) \tag{13.4}$$

$\lceil x \rceil$ is the smallest whole number not less than x; the max *keeps one empty page when nothing matches. Book: Section 13.12.1.*

Worked Example 3: 2,953 Bills in Pages

Table 13.7 Equation (13.2), Equation (13.3) and Equation (13.4) worked on the 2,953 bills. The last row is the clamp: a page number beyond the end becomes the last page.

Page size s	Page p	Offset $o=(p-1)s$	Rows shown	Grid prints
25	1	0	1–25	Showing 1–25 of 2,953 bills · page 1 of 119
25	2	25	26–50	Showing 26–50 of 2,953 bills · page 2 of 119
25	118	2,925	2,926–2,950	Showing 2926–2950 of 2,953 bills · page 118 of 119
25	119	2,950	2,951–2,953	Showing 2951–2953 of 2,953 bills · page 119 of 119
100	30	2,900	2,901–2,953	Showing 2901–2953 of 2,953 bills · page 30 of 30
25	999	clamped to page 119	2,951–2,953	Showing 2951–2953 of 2,953 bills · page 119 of 119

Fourth row of Table 13.7: $o=(119-1)\times 25=2950$*, the first row shown is* 2951*, and the last is* $\min(119\times 25,2953)=\min(2975,2953)=2953$*: the last page has three bills. Exercise 13.2 does the same for Keputih's 996 bills. Book: Section 13.12.1.*

One Page, Two Queries

Listing 13.16 `fetch_page()`: count and total first, clamp the page number, then fetch exactly one page with `LIMIT ? OFFSET ?`, both bound.

```
def fetch_page(conn, state):
    """One page of rows plus the numbers the pager and the footer need."""
    where_sql, values = where(state)
    total, charged = conn.execute(
        f"SELECT COUNT(*), COALESCE(SUM(b.payment), 0) FROM Bill b WHERE {where_sql}",
        values).fetchone()
    pages = max(1, math.ceil(total / state.size))
    state = replace(state, page=min(state.page, pages))      # page 999 becomes the last page
    rows = conn.execute(
        f"{SELECT_ROWS} WHERE {where_sql} ORDER BY {order_by(state)} LIMIT ? OFFSET ?",
        values + [state.size, (state.page - 1) * state.size]).fetchall()
    return {"state": state, "rows": rows, "total": total, "charged": charged, "pages": pages}
```

In Listing 13.16 both queries share one WHERE clause, so count and rows agree; the clamp turns `?page=999` into the last page. PHP's PDO emulates prepares for MySQL and would send `LIMIT '25'`, so the PHP grid binds limit and offset with `PDO::PARAM_INT`. Book: Section 13.12.2.

OFFSET Walks, Keyset Seeks

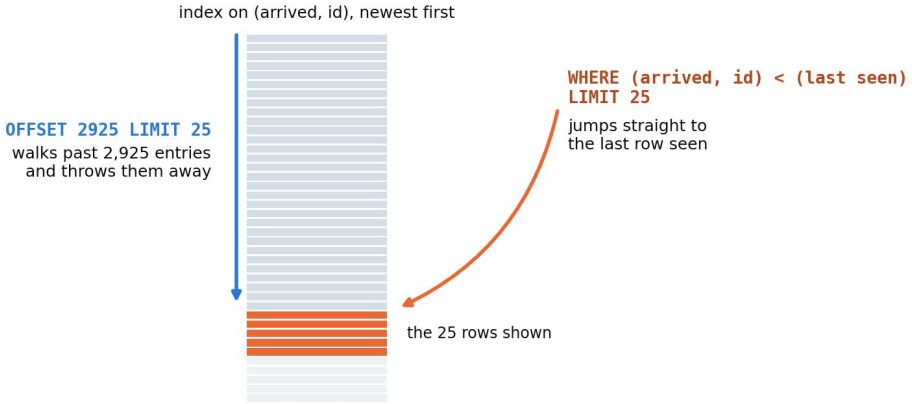


Figure 13.9 Two ways to fetch the same 25 rows from an index kept newest first. OFFSET walks past every earlier entry and discards it; keyset paging asks for the rows *after* the last one already shown and lets the index go straight there.

`LIMIT 25 OFFSET 2925` *cannot jump to row 2,926: a table has no row numbers, so the database produces the ordered rows and counts off 2,925 of them. OFFSET skips output, not work (Figure 13.9). Book: Sections 13.13.1 and 13.13.2.*

Two Cost Models and One Condition

OFFSET costs a fixed t_0 plus c per row walked past, Equation (13.5):

$$T_{\text{offset}}(p) \approx t_0 + c\,(p-1)\,s \tag{13.5}$$

Keyset paging continues after the last row shown, (a_0, i_0), with the condition Equation (13.6), and an index search costs Equation (13.7):

$$a < a_0 \vee (a = a_0 \wedge i < i_0) \tag{13.6}$$

$$T_{\text{keyset}} \approx t_0 + c' \log N \tag{13.7}$$

In words: arrived earlier, or at the very same second with a smaller id. The second clause is why the tie-breaker is mandatory here. For an ascending grid both `<` *become* `>`*. Book: Sections 13.13.1 and 13.13.2.*

The Same Rows, Written to Seek

The OR form, `WHERE (arrived < ? OR (arrived = ? AND id < ?))`, and the row value, `WHERE (arrived, id) < (?, ?)`, select the same rows as Equation (13.8):

$$a \le a_0 \wedge \left(a < a_0 \vee i < i_0 \right) \tag{13.8}$$

Proof by cases. $a < a_0$: the bracket is true. $a = a_0$: the bracket reduces to $i < i_0$. $a > a_0$: both are false.

The rows are the same; the planner differs. The leading `arrived <= ?` is a range on the first index column, which becomes a seek; SQLite reads the OR form as a scan. Book: Section 13.13.3.

Measured on the Laundry's Bills

Listing 13.17 Lines from Parts 1 and 2b of the real output of `bench/paging_bench.py` (SQLite 3.45.1). Medians in milliseconds; "same rows? yes" means the keyset page matched the OFFSET page.

```
Part 1: the real table, 2,953 bills = 119 pages, the grid's own query
  every row, to show 25 of them (what SqlDataSource paging does): 2,953 rows in 9.504 ms
   page  OFFSET  OFFSET ms  keyset ms  same rows?
      1       0      0.113      0.115  yes
      2      25      0.113      0.086  yes
     30     725      1.605      0.138  yes
     60    1475      3.124      0.085  yes
     90    2225      4.547      0.085  yes
    118    2925      6.240      0.088  yes
    119    2950      7.330      0.027  yes
Part 2b: 295,300 rows (100 copies) = 11,812 pages, with an index on (arrived, id)
  plan, keyset row:    SEARCH BigBill USING INDEX BigBill_arrived_id ((arrived,id)<(?,?))
  plan, keyset prefix: SEARCH BigBill USING INDEX BigBill_arrived_id (arrived<?)
    page   OFFSET  OFFSET ms     key-or ms    key-row ms key-prefix ms  same rows?
       1        0      0.026         0.024         0.024         0.024  yes
    8000   199975      6.327        24.636         0.023         0.023  yes
   11812   295275     14.773        34.795         0.023         0.023  yes
```

Listing 13.17, Part 1: OFFSET grows about 2.1 µs per row walked past, keyset stays near 0.1 ms, and fetching all 2,953 rows costs 9.504 ms, about 84 times one page. Part 2b, 295,300 rows with an index: OFFSET reaches 14.773 ms; the OR form scans and is worse; the row value and the prefix form SEARCH at 0.023 ms. Book: Section 13.13.4.

The Measurements, Drawn

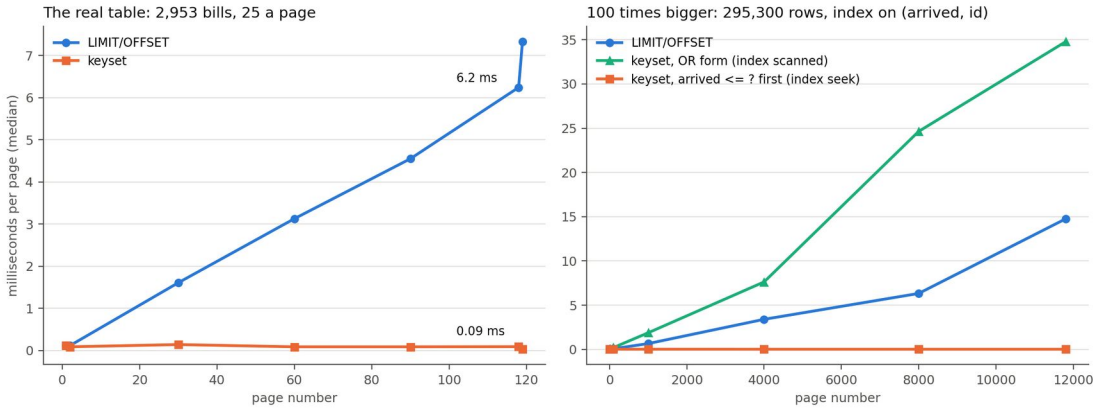


Figure 13.10 Milliseconds per page against page number, from Listing 13.17. Left: the real table, where OFFSET grows in a straight line and keyset stays flat. Right: 295,300 rows with an index, where the OR form of keyset is *slower* than OFFSET and the form of Equation (13.8) stays at 0.023 ms on every page.

> **THE KEY INSIGHT.** Keyset paging is fast only if the database can *seek* to the key. Writing the condition so that the index is used (a leading range on the first index column, or a row-value comparison) is as important as choosing keyset paging at all. Read the query plan; do not assume.

In Figure 13.10, without an index (Part 2a) every method sorts 295,300 rows per page and OFFSET climbs to 399 ms: for a big grid the index is not optional. Book: Section 13.13.4.

Which One the Grid Uses

OFFSET, rightly: its worst real page costs 6 ms, and only OFFSET gives page numbers, a *Last* link and “page 60 of 119”.

Keyset belongs to feeds, infinite scroll, very large tables, and tables that change while someone reads them. Listing 13.18 is the Flask grid’s version.

Listing 13.18 `fetch_after()` in `python/grid.py`: keyset paging with the index-friendly condition, under the same filters as the numbered pages.

```
def fetch_after(conn, state, after_arrived, after_id):
    """Keyset paging (newest first): the rows that come after the last one shown.
    'b.arrived <= ?' comes first so that the database can seek in its index."""
    where_sql, values = where(state)
    return conn.execute(
        f"{SELECT_ROWS} WHERE {where_sql} AND b.arrived <= ? AND (b.arrived < ? OR b.id < ?) "
        "ORDER BY b.arrived DESC, b.id DESC LIMIT ?",
        values + [after_arrived, after_arrived, after_id, state.size]).fetchall()
```

`python/test_keyset.py` walks all 2,953 bills by page number and by Older → links and checks that both walks agree, bill for bill. Book: Section 13.13.5.

Editing the Status in Place

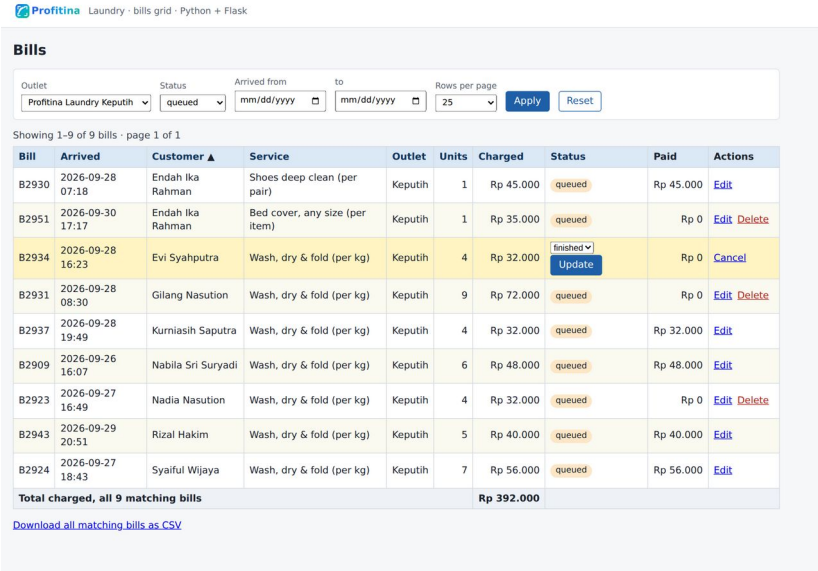


Figure 13.11 The Flask grid at `/bills?outlet=O02&status=queued&sort=customer&dir=asc&edit=B2934`. Row B2934 is drawn as a form offering its one next status; *Cancel* is a link back to the same grid without `edit`.

A bill moves `queued → finished → departed`, one step at a time; each step stamps its time and writes one StatusLog row. The Edit link in Figure 13.11 is the current grid plus `edit=B2934`: the modern `EditIndex`. Book: Section 13.14.1.

The Edit Row

Listing 13.19 The edit row in `python/templates/_grid.html`: the grid's state in hidden fields, the bill's id and *current* status, the one allowed next status, and a Cancel link.

```
      {% if r.id == edit and r.status in next_status %}
      <tr class="editing" data-id="{{ r.id }}">

<td>{{ r.id }}</td><td>{{ r.arrived[:16] }}</td><td>{{ r.customer }}</td><td>{{ r.service }}</td>
        <td>{{ r.outlet|replace("Profitina Laundry ", "") }}</td><td class="num">{{ r.unit }}</td>
        <td class="num">{{ r.payment|rupiah }}</td>
        <td>
          <form method="post" action="/bills/status">
            {{ state_fields() }}
            <input type="hidden" name="id" value="{{ r.id }}">
            <input type="hidden" name="from" value="{{ r.status }}">
            <select name="to" aria-label="New status">
              <option value="{{ next_status[r.status] }}">{{ next_status[r.status] }}</option>
            </select>
            <button type="submit">Update</button>
          </form>
        </td>
        <td class="num">{{ r.paid|rupiah }}</td>
        <td class="actions"><a data-grid href="{{ state.href() }}">Cancel</a></td>
      </tr>
```

Listing 13.19 sends the status the row had when it was drawn (`from`) as well as the new one (`to`). The next page shows why. Book: Section 13.14.1.

Worked Example 4: Two Cashiers, One Bill

Listing 13.20 `advance()` in `python/grid.py`: one legal step, conditional on the status still being `old`, with its StatusLog row in the same transaction.

```
def advance(conn, bill_id, old, new, now, actor):
    """Move a bill one step along queued -> finished -> departed. Returns a message code."""
    if not BILL_RX.match(bill_id) or NEXT_STATUS.get(old) != new:
        return "bad-request"
    stamp = "finished" if new == "finished" else "departed"
    with transaction(conn):                                  # both writes or neither
        cur = conn.execute(
            f"UPDATE Bill SET status = ?, {stamp} = ? WHERE id = ? AND status = ?",
            (new, now, bill_id, old))
        if cur.rowcount == 0:                                # gone, or changed meanwhile
            exists = conn.execute("SELECT 1 FROM Bill WHERE id = ?", (bill_id,)).fetchone()
            return "conflict" if exists else "not-found"
        conn.execute("INSERT INTO StatusLog (Bill_id, status, changed_at, Employee_id) "
                     "VALUES (?, ?, ?, ?)", (bill_id, new, now, actor))
    return new
```

*Both cashiers see B2934 queued. The first POST updates one row. The second runs the same `UPDATE … WHERE id = ? AND status = ?` with `queued`, matches **zero rows**, finds the bill exists, and returns `conflict` (Listing 13.20). This is optimistic concurrency; `BEGIN IMMEDIATE` holds SQLite's write lock from the start. Book: Section 13.14.2.*

Post/Redirect/Get

Listing 13.21 The two POST handlers of `python/app.py` and the redirect they share. `code=303` is *See Other*: "fetch the result with GET".

```
def back_to_grid(code, bill_id):
    """Post/Redirect/Get: rebuild the grid's address from validated values only."""
    state = current_state(request.form)
    bill = bill_id if grid.BILL_RX.match(bill_id) else ""
    pairs = state.params() + [("msg", code), ("bill", bill)]
    return redirect("/bills?" + urlencode(pairs), code=303)


@app.post("/bills/status")
def change_status():
    bill_id = request.form.get("id", "")
    code = grid.advance(db(), bill_id, request.form.get("from", ""),
                        request.form.get("to", ""), now_text(), ACTOR)
    return back_to_grid(code, bill_id)


@app.post("/bills/delete")
def delete_bill():
    bill_id = request.form.get("id", "")
    return back_to_grid(grid.delete(db(), bill_id), bill_id)
```

In Listing 13.21 the address carries a message code (`finished`, `deleted`, `conflict`, `refused-paid`, …) and a bill id, never a sentence, so a forged link cannot make the grid say anything. Reload repeats the harmless GET, not the change. Book: Section 13.14.3.

Asking Without a Script

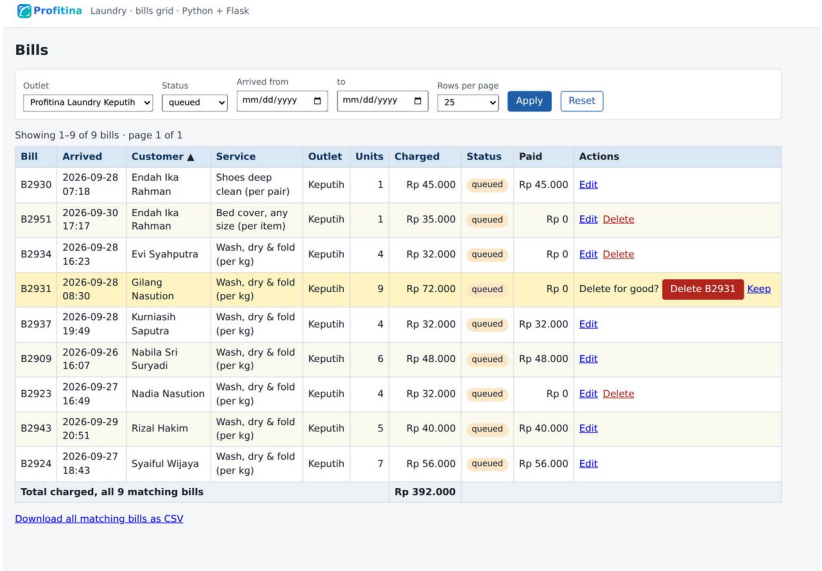


Figure 13.12 The confirmation row for B2931, drawn by the server. *Delete B2931* submits a POST form; *Keep* is a link back to the same grid. No script is involved.

The rule: ***a bill may be deleted only while it is queued and has no payment*** *(eight bills). The Delete link goes to the grid plus* `delete=B2931`*, and the server draws the question as a row with a real form (Figure 13.12); it works with scripts off and for a screen reader. Book: Sections 13.15.1 and 13.15.2.*

The Delete, Checked in One Transaction

Listing 13.22 `delete()` in `python/grid.py`: check the bill's status and payments and delete its log row and the bill, all inside one transaction that holds the write lock from the start.

```
def delete(conn, bill_id):
    """Delete a bill typed by mistake: only while queued and unpaid. Returns a message code."""
    if not BILL_RX.match(bill_id):
        return "bad-request"
    with transaction(conn):
        row = conn.execute(
            "SELECT status, (SELECT COUNT(*) FROM Payment p WHERE p.Bill_id = b.id) "
            "FROM Bill b WHERE b.id = ?", (bill_id,)).fetchone()
        if row is None:
            return "not-found"
        if row[0] != "queued":
            return "refused-status"
        if row[1] > 0:
            return "refused-paid"
        conn.execute("DELETE FROM StatusLog WHERE Bill_id = ?", (bill_id,))
        conn.execute("DELETE FROM Bill WHERE id = ? AND status = 'queued'", (bill_id,))
    return "deleted"
```

*Checking and deleting in separate steps is a **check-then-act race**: a payment could arrive in between. Listing 13.22 deletes the StatusLog row first because `ON DELETE RESTRICT` would refuse the bill while it exists. Book: Section 13.15.3.*

A CSV That Is Safe to Open

Listing 13.23 `csv_cell()` and `csv_lines()` in `python/grid.py`: neutralize formulas, quote when needed, write the BOM once, end every line with CR LF.

```
def csv_cell(value):
    """One CSV field: quoted when needed, and never read as a formula by a spreadsheet."""
    text = str(value)
    if text[:1] in ("=", "+", "-", "@", "\t", "\r"):
        text = "'" + text
    if any(ch in text for ch in ',"\r\n'):
        text = '"' + text.replace('"', '""') + '"'
    return text


CSV_HEADER = ("Bill", "Arrived", "Customer", "Service", "Outlet", "Units", "Charged", "Paid", "Status")


def csv_lines(rows):
    yield "\ufeff" + ",".join(CSV_HEADER) + "\r\n"          # the BOM tells Excel it is UTF-8
    for r in rows:
        yield ",".join(csv_cell(v) for v in (
            r["id"], r["arrived"], r["customer"], r["service"], r["outlet"], r["unit"],
            int(r["payment"]), int(r["paid"]), r["status"])) + "\r\n"
```

Listing 13.23: RFC 4180 quoting and CR LF; the BOM `EF BB BF` stops Excel garbling Désirée; a leading `=`, `+`, `-` or `@` gets an apostrophe (CSV injection). Same filters and order as the grid, no paging; the file holds personal data under UU PDP 27/2022. Book: Section 13.16.

Razor Pages: Refusing Cross-Site POSTs

Listing 13.24 The middleware in `Program.cs` that refuses a POST from another site, using the `Sec-Fetch-Site` and `Origin` headers.

```
// A POST from another site's page is refused (a cheap guard against CSRF; the
// antiforgery token that Razor Pages checks on every POST is the second guard).
app.Use(async (context, next) =>
{
    var req = context.Request;
    if (HttpMethods.IsPost(req.Method))
    {
        string? site = req.Headers["Sec-Fetch-Site"].FirstOrDefault();
        string? origin = req.Headers.Origin.FirstOrDefault();
        if ((site is not null && site is not ("same-origin" or "none"))
            || (origin is not null && origin != $"{req.Scheme}://{req.Host}"))
        {
            context.Response.StatusCode = StatusCodes.Status403Forbidden;
            await context.Response.WriteAsync("Forbidden");
            return;
        }
    }
    await next();
});
```

Listing 13.24 answers a form on `evil.example` with 403: a cheap guard against CSRF. The antiforgery token that Razor Pages checks on every POST is the second guard; a POST without it gets 400. NuGet was blocked on the build machine, so the grid runs on an in-memory store; `dotnet add package Microsoft.Data.Sqlite` switches it to SQLite. Book: Sections 13.17.1 and 13.17.2.

Razor Pages: The Page Model

Listing 13.25 `OnGet` and `OnPostStatus` in `Pages/Bills.cshtml.cs`: fill the properties the template reads (or return only the grid when `partial=1`), or change a bill.

```
    public IActionResult OnGet()
    {
        Grid = store.Page(ReadState(Query));
        Edit = Query("edit");
        Confirm = Query("delete");
        Message = Text.Message(Query("msg"), Query("bill"));
        return Query("partial") == "1" ? Partial("_Grid", this) : Page();
    }

    public IActionResult OnPostStatus()
    {
        string id = Form("id");
        string code = store.Advance(id, Form("from"), Form("to"), Clock.Now(), Clock.Actor);
        return BackToGrid(code, id);
    }
```

Compare Listing 13.25 with `Page_Load`: no `IsPostBack`, no binding step, no view state. `@page "/bills/{handler?}"` maps `POST /bills/status` to `OnPostStatus`, and `BackToGrid` answers 303. Razor encodes every `@value`. Book: Sections 13.17.3 and 13.17.4.

The Razor Pages Grid Running

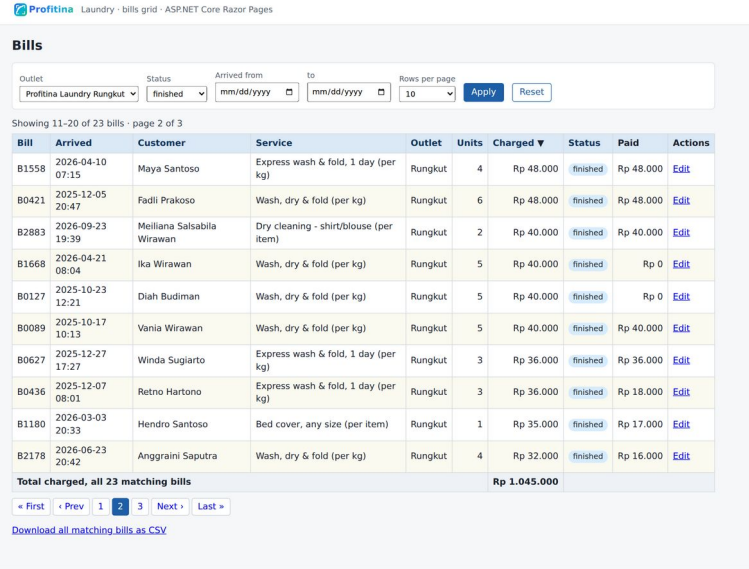


Figure 13.13 The Razor Pages grid at `/bills?outlet=O01&status=finished&sort=payment&dir=desc&size=10&page=2`: rows 11 to 20 of 23, with the full pager and a footer total over all 23 bills.

Figure 13.13 is filtered, sorted and paged, and the whole state is in the address bar: bookmark it, share it, reload it. Book: Section 13.17.5.

Progressive Enhancement with `fetch()`

Listing 13.26 `load()` in `static/grid-enhance.js`: fetch the grid fragment with `partial=1`, swap it in, keep the address bar in step, and fall back to a full page load on any failure.

```
    return fetch(withPartial(url), { headers: { "Accept": "text/html" } })
      .then(function (res) {
        if (!res.ok) { throw new Error("HTTP " + res.status); }
        return res.text();
      })
      .then(function (fragment) {
        area.outerHTML = fragment;            // the server sent <div id="grid">...</div>
        area = document.getElementById("grid");
        if (push) { history.pushState(null, "", url); }
        syncFilters(url);
      })
      .catch(function () {
        location.href = url;                  // anything odd: fall back to a full page load
      });
  }
```

The grid works completely with links and forms; Listing 13.26 only makes it better. `pushState` *keeps a real, shareable address;* `popstate` *handles Back; a Delete link gets* `confirm()`*; anything odd becomes an ordinary page load. Playwright checked eight behaviors on all four grids. Book: Section 13.21.*

One Set of Checks for Four Servers

Listing 13.27 Lines from the real output of `tests/run_checks.sh java`: 44 checks of reading, injection, the status workflow, deleting, cross-site POSTs and the database afterwards.

```
PASS  an injected sort column is ignored (default order)
PASS  an injected outlet is ignored (no filter)
PASS  status queued -> finished answers 303 See Other
PASS  the redirect keeps the grid's state and names the result
PASS  the message is shown and the bill moved
PASS  the same stale edit again is a conflict
PASS  skipping a step is refused
PASS  an unknown bill is reported
PASS  a queued but partly paid bill is not deleted
PASS  a POST from another origin is refused with 403
PASS  a cross-site POST (Fetch Metadata) is refused with 403
PASS  database: B2953 finished at the shop clock
PASS  database: one StatusLog row written in the same transaction
PASS  database: B2951 and its log are gone
PASS  database: B2950 untouched by the refused cross-site POST

44 passed, 0 failed
```

The checks in Listing 13.27 compute the right answers with their own SQL from the pristine database, so a server passes only by agreeing with an independent calculation. PHP, Java and Flask pass 44 of 44; Razor Pages 40 of 40 (no database file); the Web Forms pages were parsed by Roslyn, not run. Book: Section 13.22.

Profitina in Practice

A history is a grid over a growing table: the weekly score dashboard is where an index that serves the grid's order keeps week 100 as quick as week 1.

Export is a feature, and personal data comes with it: the CSV export follows the screen's filters, a stable order and no live formulas; UU PDP duties are features.

A feed is keyset paging's home: posts arrive at the top all the time, and a reader must neither see one twice nor miss one.

Two databases, one SQL, and **instant, but never dependent on the trick:** a post that fails to send is restored.

Profitina runs FastAPI with PostgreSQL at `app.profitina.com`; the runnable case study stays Profitina Laundry. Book: Section 13.23.

Common Mistakes

`"ORDER BY " + sort`, or `ORDER BY ?` that silently sorts by a constant; sorting a paged query without a unique tie-breaker.

Paging in memory; keyset with only the OR form; `arrived <= '2026-09-30'` for “until the 30th”; `LIMIT '25'` from PDO with MySQL.

A key read from displayed text; trusting your own database (second-order injection); relying on `confirm()` or hidden links.

An update that ignores what the user saw; change and audit row in separate transactions; answering a POST with a page; redirecting to an address from the form; message text in the address.

A CSV without quoting, BOM or formula guard; an “enhancement” the grid cannot work without.

The book lists each mistake with its fix. Book: Section 13.24.

Summary: Five Things to Remember

1. A data grid does seven jobs (Table 13.1); GridView did them with postbacks and view state, paging in memory, with its state invisible to the address bar.

2. The modern grid is links and forms: reads are GETs with the whole state in the address; changes are POSTs answered with 303 (Figure 13.8).

3. Filter values are bound; sort keys come from a white list, ending with a unique tie-breaker; “to” means before the next midnight (Equation (13.1)).

4. Paging is arithmetic (Equation (13.2), Equation (13.3)); OFFSET walks, keyset seeks when its condition lets the index be searched (Equation (13.8)).

5. Edits are conditional and transactional, deletes are narrow and confirmed by the server, exports are quoted and formula-safe; one script enhances all four grids without being needed.

Book: Section 13.25; Exercises 13.1–13.12 with solutions in Sections 13.26 and 13.27.

Next Time

Chapter 14 — Validation and State Management. A check in the browser is a courtesy; a check on the server is a law.

The six Web Forms validators, then data annotations in Razor Pages; view state, cookies, session, TempData and the cache, chosen by *who may see this?* and *how long must it live?*

Before then: work Exercises 13.2, 13.4, 13.6 and 13.10, and turn off JavaScript and use the grid (Exercise 13.11).

Today's `Clock.Actor` and `LAUNDRY_ACTOR` stand in for a login; Chapter 14 begins to replace them. Book: Chapter 14.

Lecture 14 — Validation and State Management

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 14

Lecturer: Irfan Subakti — yifana@gmail.com

Today we stop bad data at the door and make a forgetful protocol remember, at the front desk of Profitina Laundry. Book: Chapter 14, Sections 14.1–14.15.

What This Meeting Covers

1. **Why validate, and where** — three kinds of bad data; the browser helps, the server decides.
2. **The Web Forms validators** — required, compare, range, regular expression, custom, summary; the Laundry form; user controls.
3. **Validation in Razor Pages** — data annotations, a custom attribute for Indonesian mobile numbers, `ModelState`, unobtrusive browser checks.
4. **HTTP forgets** — the map of state mechanisms, chosen by scope and lifetime.
5. **Client and server state** — view state, hidden fields, query strings, cookies; session, TempData, application state, the cache.
6. **Four tracks, security and UU PDP** — one front desk in Razor Pages, PHP, Java and Flask, and where personal data may live.

Validation is about one request; state management is about many requests. They meet constantly, because every piece of state that comes back from the browser is input. Book: Section 14.1.

Where We Left Off

Chapter 13 filled a grid with the Laundry's bills. Earlier chapters left two promises open: *what if the person typing makes a mistake, or lies?* and *how did a Web Forms text box remember its value, when HTTP remembers nothing?*

Validation decides, for every value that arrives, whether it may enter the system. **State management** carries information from one request to the next.

> **THE KEY INSIGHT.** Everything that comes from the browser is input — form fields, hidden fields, query strings, cookies, headers — and all input is validated on the server, every time. State that the browser keeps is a *suggestion*; state that the server keeps is a *fact*.

The outlet cookie, the hidden customer id and the number in the query string all look like memory, and all are input. Book: Section 14.1.

Three Kinds of Bad Data

Table 14.1 Three kinds of bad data, with examples from the Profitina Laundry front desk.

Kind	What it means	At the Laundry counter
Useless	The value is missing or has the wrong shape, so nothing can be done with it	An empty name; a phone number `021-555-0199` that cannot receive a WhatsApp message saying the laundry is ready; `2.5` bed covers
Unauthenticated	The value claims something that nobody has checked	A hidden field that says the bill belongs to customer `C0001` when the cashier opened the draft for `C0257`; a cookie that says the counter is outlet `O09`, which does not exist
Contradictory	Each value is fine alone, but together they cannot all be true	Two passwords that do not match; a date of birth in 2031; two lines of one draft on the same rack `B09`; a rack that already holds another customer's clothes

*In **Table 14.1** only useless values can be recognized alone; the other two need the server's knowledge. The browser can help but never decide. Book: Section 14.2.1.*

Four Places a Value Passes

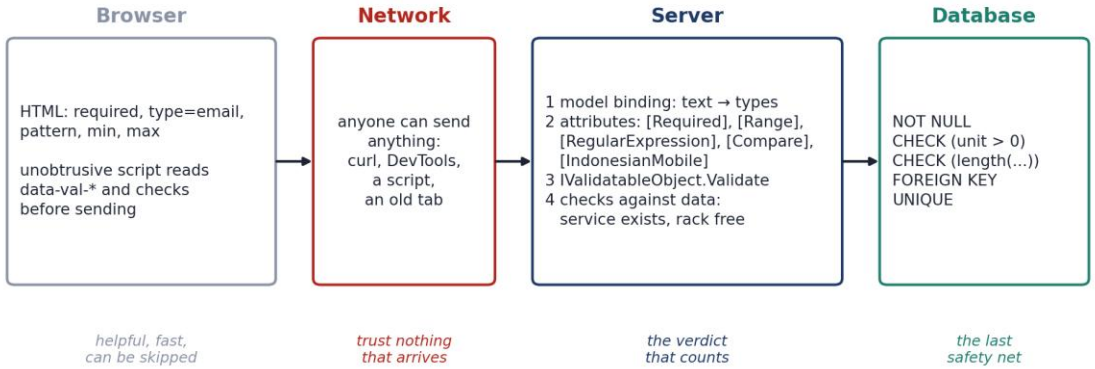


Figure 14.1 The four places a value passes on its way into the Laundry database. The browser's checks are fast and friendly but can be skipped; between the browser and the server anyone can send anything; the server's checks are the ones that count; the database's constraints are the last safety net.

*Read **Figure 14.1** left to right:* `curl -d "Input.Name=" http://127.0.0.1:9401/customers/new` *sends a form no browser check ever saw. The database's constraints are a safety net, not a user interface. Book: Section 14.2.2.*

Client Side and Server Side

> **DEFINITION. Client-side validation** checks input in the browser before it is sent, to give fast feedback. **Server-side validation** checks input on the server after it arrives, to protect the application. The first is a convenience for honest users; the second is a control against everybody. An application needs the second always and the first usually.

In the browser: JavaScript, or the HTML attributes browsers understand natively — `required`, `type="email"`, `pattern`, `min`, `max`.

On the server: after the request has arrived and before anything is stored — the verdict that counts.

JavaScript can be switched off, the page edited in developer tools, or the browser skipped altogether. Book: Section 14.2.2.

A Check Written by Hand

Listing 14.1 The script and the form tag of `01FirstNameFilling.aspx` (Web Forms, not run here): `onsubmit` calls a function that refuses an empty first name.

```
    <script>
        function checkForm() {
            var x = document.forms["myForm"]["fname"].value;
            if (x == "") {
                alert("First name has to be filled!");
                return false;
            }
            return true;
        }
    </script>
    <form id="myForm" runat="server"
        action="01Done.html" onsubmit="return checkForm()"
        method="get">
```

*In **Listing 14.1**, a `false` from `checkForm()` cancels the submission. Nothing on the server checks the first name: `01Done.html` says Your data has been saved! to anybody. Book: Section 14.2.3.*

The Browser Refuses

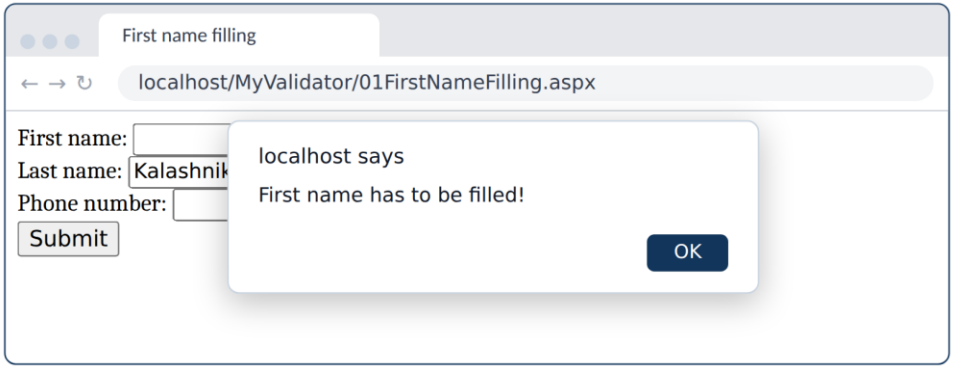


Figure 14.2 The hand-written check of **Listing 14.1** at work: the first name is empty, so `checkForm()` shows an alert and returns `false`, and the form is not submitted.

***Figure 14.2**: the browser's own dialog, and nothing is sent. Every `.aspx` example in this chapter ran on Windows; the book's build machine is Linux. Book: Section 14.2.3.*

State in the Address

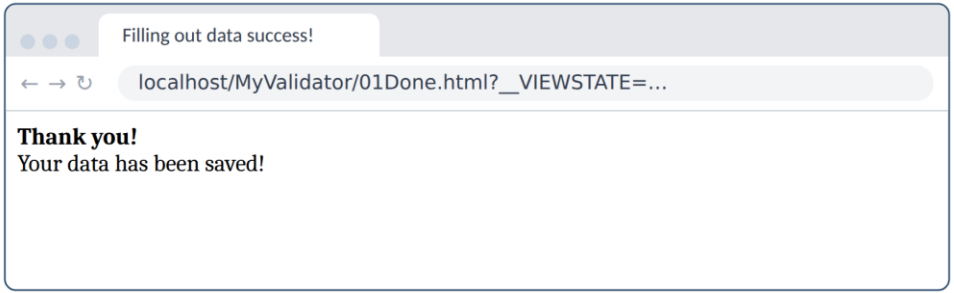


Figure 14.3 The thank-you page after a successful submission. The address begins `01Done.html?__VIEWSTATE=…`: because the form used GET, every field went into the URL — including the hidden `__VIEWSTATE` field that Web Forms added on its own.

*The form used GET, so every field went into the address (**Figure 14.3**), including `__VIEWSTATE`, a hidden field the page never declared. The state of the page travelled with the request. Book: Section 14.2.3.*

Worked Example 1: No Browser at All

`tests/http_check.py` keeps cookies and hidden fields but runs no script. It sent a completely wrong customer form to the Razor Pages front desk:

```
POST /customers/new (bad)   422, 7 messages:
    - Enter the customer's name.
    - Choose female or male.
    - Enter a date of birth in the past, at most 120 years ago.
    - Enter an Indonesian mobile number such as 0812-3456-7890 or +62 812 3456 7890.
    - Enter a valid e-mail address, for example rina@example.com.
    - An online password needs 8 to 64 characters.
    - The two passwords do not match.
  place kept               Surabaya
  password sent back       no
```

> **Client-side validation is a user-interface feature, not a security feature.** Write it so that honest people fix their mistakes quickly. Never write it *instead of* the server check.

422 Unprocessable Content: "understood, and refused". The place of birth came back filled in; the password did not. Book: Section 14.2.4.

The Six Validator Controls

Table 14.2 The six validator controls of Web Forms and the Razor Pages tool that does the same job.

Control	What it checks	Key properties	In Razor Pages
`RequiredFieldValidator`	The field is not empty (or differs from `InitialValue`)	`ControlToValidate`, `InitialValue`	`[Required]`
`CompareValidator`	Equal to another field, compared with a constant, or of a data type	`ControlToCompare` or `ValueToCompare`, `Operator`, `Type`	`[Compare]`; a typed property; a custom check
`RangeValidator`	Between a minimum and a maximum of a given type	`MinimumValue`, `MaximumValue`, `Type`	`[Range]`
`RegularExpressionValidator`	Matches a pattern	`ValidationExpression`	`[RegularExpression]`, `[EmailAddress]`, `[StringLength]`
`CustomValidator`	Any rule you write, in the browser and on the server	`ClientValidationFunction`, `OnServerValidate`	A custom `ValidationAttribute`, `IValidatableObject`, or `ModelState.AddModelError`
`ValidationSummary`	Nothing itself: it lists the messages of the others	`HeaderText`, `DisplayMode`, `ShowMessageBox`	`asp-validation-summary`

*Validators in **Table 14.2** only report into `Page.IsValid`. Book: Section 14.3.1.*

`RequiredFieldValidator`

Listing 14.2 From `02UserIDRequired.aspx` (Web Forms, not run here): a `RequiredFieldValidator` whose own text is an asterisk and whose `ErrorMessage` goes to a `ValidationSummary`.

```
            <td><input type="text" runat="server" id="txtID" />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                                            runat="server"
                                            ControlToValidate="txtID"
                                            ErrorMessage="User ID is required.">
                    *
                </asp:RequiredFieldValidator>
    <asp:ValidationSummary ID="ValidationSummary1" runat="server"
                           HeaderText="There were errors on the page:" />
```

*In **Listing 14.2** the asterisk between the tags is the validator's `Text`, shown next to the field; the `ErrorMessage` is collected by the summary. A validator can guard a plain `<input runat="server">` too. Book: Section 14.3.2.*

An Empty User Id

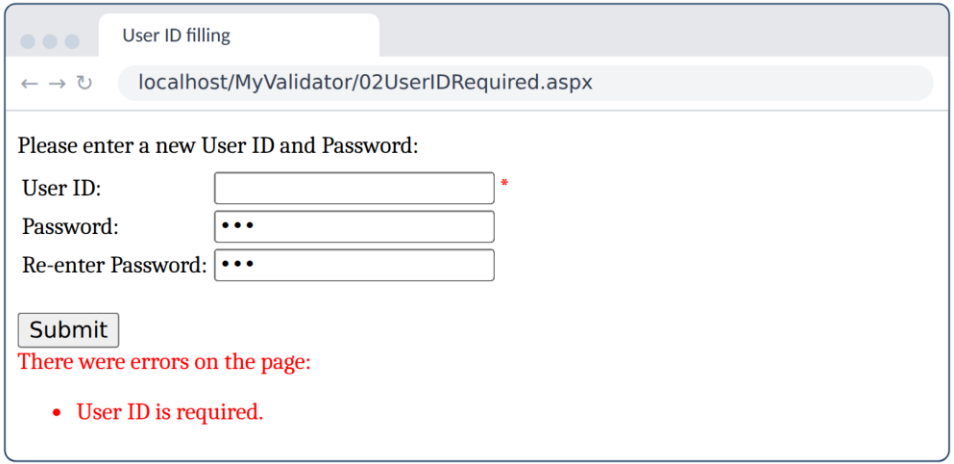


Figure 14.4 Listing 14.2 after **Submit** with an empty user id: an asterisk beside the field, and the full message under *There were errors on the page:*.

Figure 14.4*: two places for one problem, the mark beside the field and the sentence in the list. Book: Section 14.3.2.*

`CompareValidator`: Three Jobs

Listing 14.3 From `03PasswordsNotMatch.aspx` (Web Forms, not run here): the second password must equal the first.

```
        <asp:CompareValidator ID="CompareValidator1" runat="server"
            ControlToValidate="txtRePassword"
            ControlToCompare="txtPassword"
            ErrorMessage="Passwords do not match." />
```

Listing 14.4 From `05GreaterThan18.aspx` (Web Forms, not run here): a comparison with a constant, as integers.

```
            <asp:CompareValidator id="CompareValidator1" runat="server"
                Operator="GreaterThan" ValueToCompare="18"
                ControlToValidate="TextBox1"
                ErrorMessage="You must be older than 18"
                Type="Integer">
            </asp:CompareValidator>
```

Listing 14.3 *uses* `ControlToCompare`*;* `Operator="DataTypeCheck"` *checks a type;* ***Listing 14.4*** *uses* `ValueToCompare`*. Without* `Type="Integer"`*,* `"9"` *is greater than* `"18"`*. The same trap waits in* `RangeValidator`*, and every validator but the required one accepts an empty field. Book: Sections 14.3.3 and 14.3.4.*

Passwords That Differ

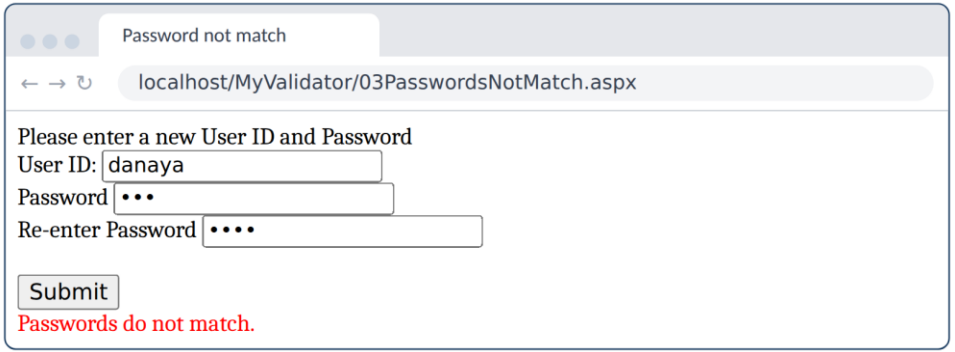


Figure 14.5 Listing 14.3 with a three-character password and a four-character repetition: *Passwords do not match.*

***Figure 14.5**: the validator guards the second box and the operator defaults to `Equal`. A `RangeValidator` from 30 to 40 answers 17 the same way. Book: Sections 14.3.3–14.3.4 and Figure 14.6.*

`RegularExpressionValidator`

Listing 14.5 The password box of `08PasswordsRegEx.aspx` (Web Forms, not run here): two regular expressions, and the comment that explains the required check beside them.

```
            <asp:TextBox type="password" ID="txtPassword" runat="server"></asp:TextBox>
            <asp:RegularExpressionValidator ID="RegularExpressionValidator3"
                runat="server"
                display="Dynamic"
                ControlToValidate="txtPassword"
                ErrorMessage="Password must be 4-12 nonblank characters."
                ValidationExpression="[^\s]{4,12}" />
            <asp:RegularExpressionValidator ID="RegularExpressionValidator2"
                runat="server"
                display="Dynamic"
                ControlToValidate="txtPassword"
                ErrorMessage="Password must contain one of @#$%^&*/."
                ValidationExpression=".*[@#$%^&*/].*" />
            <!-- If TextBox is empty the RegularExpressionValidator doesn't work.
                 So, we need to use the RequiredFieldValidator, too. -->
```

*In **Listing 14.5**, `[^\s]{4,12}` is "four to twelve non-blank characters" and `.*[@#$%^&*/].*` "one of these symbols somewhere". The pattern must match the whole value. Regular expressions check shape, not truth: `1234@1234.com` does not exist. Book: Section 14.3.5.*

Two Validators Speak

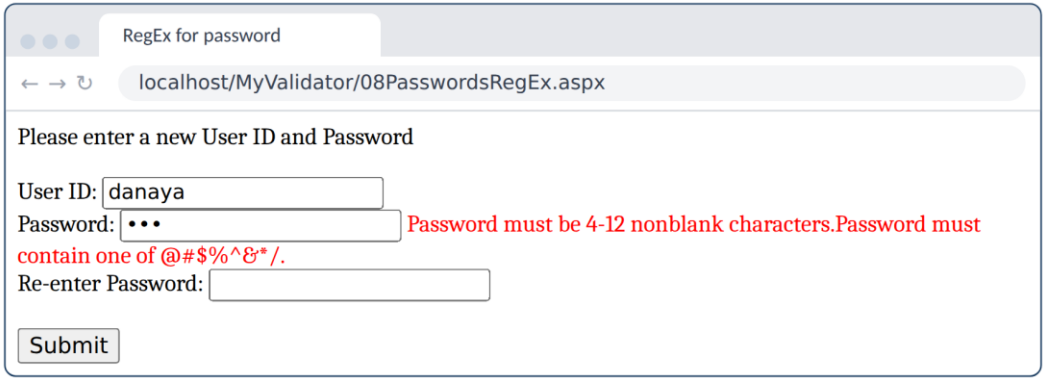


Figure 14.6 Listing 14.5 with the password `abc`: it is too short and it contains none of the required symbols, so two validators speak.

Figure 14.6: `Display="Dynamic"` *leaves no blank space for the required-field message, which has nothing to say. Book: Section 14.3.5.*

`CustomValidator` on the Server

Listing 14.6 `ValidateNumber` and its validator in `11ServerValidation.aspx` (Web Forms, not run here): the rule “divisible by 5”, run on the server.

```
        void ValidateNumber(object source, ServerValidateEventArgs args) {
            try {
                int num = int.Parse(args.Value);
                args.IsValid = ((num % 5) == 0);
            } catch (Exception ex) {
                args.IsValid = false;
            }
        }
            <asp:CustomValidator id="CustomValidator1"
                runat="server" ControlToValidate="TextBox1"
                ErrorMessage="Number must be divisible by 5"
                OnServerValidate="ValidateNumber">
```

*The browser-only version, `ClientValidationFunction`, lets 13 through when the script is skipped: `Page.IsValid` stays `true`. **Listing 14.6** refuses 3 and `abc` on the server. The right design sets both attributes, with the same rule written twice. Book: Section 14.3.6 and Figures 14.8–14.9.*

`ValidationSummary`

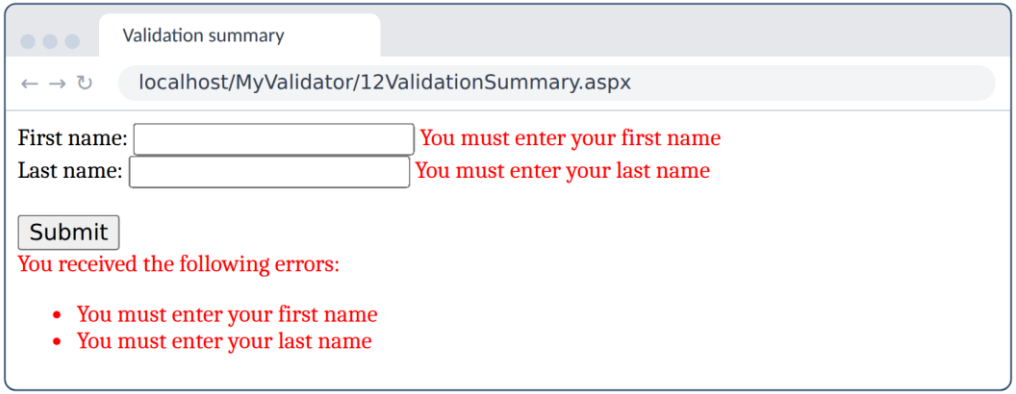


Figure 14.7 `12ValidationSummary.aspx` after an empty submit: each validator shows its message next to its field, and the summary repeats both under *You received the following errors:*.

***Figure 14.7**: `DisplayMode` is `BulletList`, `List` or `SingleParagraph`; `ShowMessageBox="true"` also shows an alert. The summary says that; the field messages say what. Book: Section 14.3.7.*

The Laundry Form, the Web Forms Way

Listing 14.7 The phone rule of `CustomerNew.aspx` (Web Forms, not run here): the browser half in JavaScript, and a `CustomValidator` that names both halves.

```
        function validatePhone(source, args) {
            var s = args.Value.replace(/[\s\-.()]/g, "");
            if (s.indexOf("+62") === 0) s = "0" + s.slice(3);
            else if (s.indexOf("62") === 0) s = "0" + s.slice(2);
            args.IsValid = /^08[1-9][0-9]{7,10}$/.test(s);
        }
        <asp:Label runat="server" AssociatedControlID="txtPhone" Text="Mobile phone" />
        <asp:TextBox ID="txtPhone" runat="server" />
        <asp:RequiredFieldValidator runat="server" ControlToValidate="txtPhone"
            ErrorMessage="Enter a mobile phone number." Text="*" Display="Dynamic" />
        <asp:CustomValidator runat="server" ControlToValidate="txtPhone"
            ClientValidationFunction="validatePhone" OnServerValidate="PhoneValidate"
            ErrorMessage="Enter an Indonesian mobile number such as 0812-3456-7890 or +62 812 3456 7890."
            Text="*" />
```

*The whole form in **Listing 14.7** uses required, range (`Type="Date"`), custom, regular-expression and compare validators, each showing an asterisk, with the summary at the top. The password is optional: no required validator. Book: Section 14.3.8.*

The Code Behind the Form

Listing 14.8 From `CustomerNew.aspx.cs` (Web Forms, not run here): the server half of the phone rule, and nothing stored unless `Page.IsValid` is `true`.

```
    // Server half of the CustomValidator. It runs even if the browser skipped the script.
    protected void PhoneValidate(object source, ServerValidateEventArgs args) {
        string s = Regex.Replace(args.Value, @"[\s\-.()]", "");
        if (s.StartsWith("+62")) s = "0" + s.Substring(3);
        else if (s.StartsWith("62")) s = "0" + s.Substring(2);
        args.IsValid = Regex.IsMatch(s, "^08[1-9][0-9]{7,10}$");
    }

    protected void Register_Click(object sender, EventArgs e) {
        if (!Page.IsValid) {          // never skip this line: it is the real check
            return;
        }
        string id = SaveCustomer();
        Session["Flash"] = "Customer " + id + " registered.";
        Response.Redirect("BillDraft.aspx?customer=" + id);
    }
```

*Without `if (!Page.IsValid) return;` in **Listing 14.8**, the page would show the red asterisks and store the customer. `Page_Load` sets the date range on every request, because "before today" changes daily. Book: Section 14.3.8.*

Worked Example 2: A Control That Remembers

Listing 14.9 `OldRate` and `Select` in `MoneyField.ascx.cs` (Web Forms, not run here): the rate of the current currency is kept in `ViewState`.

```
    private double OldRate {
        get {
            return
                ViewState["rate"] == null ? 1 : (double)ViewState["rate"];
        }
        set {
            ViewState["rate"] = value;
        }
    }
    public void Select(object sender, EventArgs arg) {
        try {
            double val = Convert.ToDouble(amount.Text);
            double newRate = Convert.ToDouble(currency.SelectedItem.Value);
            double newVal = val / OldRate * newRate;
            amount.Text = newVal.ToString("F2");
            OldRate = newRate;
```

100 in Euro (rate 1.0), choose Pounds (0.62): $100/1 \times 0.62 = 62$, *and* `OldRate` *becomes 0.62. Choose Dollars (0.88):* $62/0.62 \times 0.88 = 88$. ***Listing 14.9*** *is right only because* `ViewState["rate"]` *remembered the pounds. Book: Section 14.3.9.*

Two Money Fields on One Page

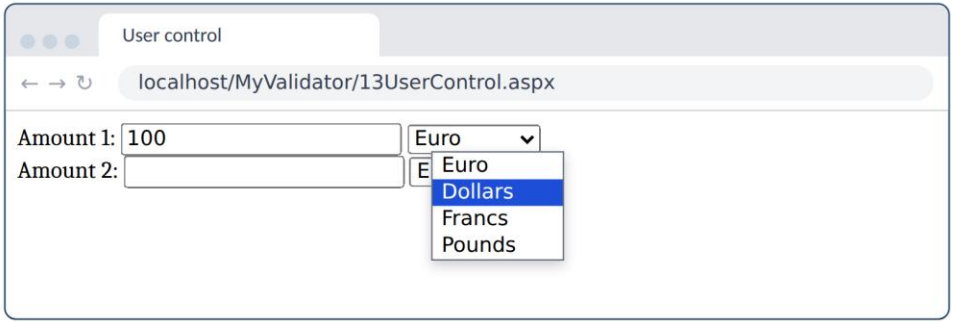


Figure 14.8 `13UserControl.aspx` in the browser: two independent money fields; the first is being switched from Euro to another currency.

*In **Figure 14.8** each instance keeps its own slice of view state, keyed by its `ID`. In ASP.NET Core the reusable fragment is a partial view, a tag helper or a view component, with no hidden view state. Book: Section 14.3.9.*

Rules Written as Attributes

Listing 14.10 From `CustomerInput.cs`: rules as data annotations on the model the form posts.

```
public class CustomerInput : IValidatableObject
{
    [Required(ErrorMessage = "Enter the customer's name.")]
    [StringLength(50, ErrorMessage = "The name can have at most 50 characters.")]
    public string? Name { get; set; }
    [Required(ErrorMessage = "Enter a mobile phone number.")]
    [IndonesianMobile]
    public string? Phone { get; set; }
    [Compare(nameof(Password), ErrorMessage = "The two passwords do not match.")]
    [DataType(DataType.Password)]
    public string? ConfirmPassword { get; set; }
```

Listing 14.10 *translates the Web Forms page almost word for word:* `[Required]`*,* `[StringLength]`*,* `[RegularExpression]`*,* `[Compare(nameof(Password))]`*;* `[IndonesianMobile]` *is our own. The Razor Pages project was run on .NET 10 with no NuGet package. Book: Sections 14.4.1 and 14.4.2.*

Four Details, and a Rule About Today

Listing 14.11 `Validate` in `CustomerInput.cs`: the `IValidatableObject` rule that needs today's date, a server-side `CustomValidator`.

```
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        var today = DateOnly.FromDateTime(DateTime.Now);
        if (Birthdate is { } born && (born >= today || born < today.AddYears(-120)))
        {
            yield return new ValidationResult(
                "Enter a date of birth in the past, at most 120 years ago.",
                new[] { nameof(Birthdate) });
        }
    }
```

Nullable properties: an empty date is `null` and `[Required]` speaks; a `DateOnly` would silently become `0001-01-01`.

Spaces are empty, and **`null` passes** every attribute but `[Required]`, so the optional password works.

Listing 14.11 *names the property it belongs to, so the message appears next to the date. Book: Section 14.4.2.*

Worked Example 3: Indonesian Mobile Numbers

Table 14.3 The Indonesian mobile rule worked by hand on six inputs, one step per column.

Typed	After removing ␣ - . ( )	After the prefix rule	Pattern `08[1-9]` + 7 to 10 digits	Stored as
`0812-3456-7890`	`081234567890`	`081234567890`	12 digits, matches	`0812-3456-7890`
`+62 812-3456-7890`	`+6281234567890`	`081234567890`	12 digits, matches	`0812-3456-7890`
`62 896 0000 9434`	`6289600009434`	`089600009434`	12 digits, matches	`0896-0000-9434`
`021-555-0199`	`0215550199`	`0215550199`	begins `02`: a Jakarta landline, refused	—
`0800-1-234-567`	`08001234567`	`08001234567`	third digit 0: toll-free, refused	—
`08123`	`08123`	`08123`	only 5 digits, refused	—

***Table 14.3**: strip separators, `+62`/`62` becomes `0`, then `08[1-9]` and 7–10 digits. Book: Section 14.4.3.*

The Rule in One Class

Listing 14.12 The pattern and `Normalize` in `Phone.cs`: what was typed becomes ten to thirteen digits, or `null`.

```
    // After cleaning: 08, then 1-9, then 7 to 10 more digits (10 to 13 digits in all).
    [GeneratedRegex("^08[1-9][0-9]{7,10}$")]
    private static partial Regex MobileDigits();
    /// <summary>"+62 812-3456-7890" -> "081234567890"; null when it is not a mobile number.</summary>
    public static string? Normalize(string? raw)
    {
        if (raw is null) return null;
        string s = Regex.Replace(raw, @"[\s\-.()]", "");
        if (s.StartsWith("+62")) s = "0" + s[3..];
        else if (s.StartsWith("62")) s = "0" + s[2..];
        return MobileDigits().IsMatch(s) ? s : null;
    }
```

*`[GeneratedRegex]` turns the pattern into C# at build time, so a mistake in it breaks the build. **Listing 14.12** returns the digits, not `true`/`false`: validation and normalization are one piece of work. `Format` stores `0812-3456-7890`. Book: Section 14.4.3.*

A Custom Validation Attribute

Listing 14.13 `IndonesianMobileAttribute`: `IsValid` runs on the server; `AddValidation` writes two `data-val-*` attributes so that the browser can run the same rule.

```
[AttributeUsage(AttributeTargets.Property)]
public sealed class IndonesianMobileAttribute : ValidationAttribute, IClientModelValidator
{
    public IndonesianMobileAttribute() =>
        ErrorMessage = "Enter an Indonesian mobile number such as 0812-3456-7890 or +62 812 3456 7890.";

    public override bool IsValid(object? value) =>
        value is null || (value is string s && Phone.Normalize(s) is not null);

    public void AddValidation(ClientModelValidationContext context)
    {
        context.Attributes.TryAdd("data-val", "true");
        context.Attributes.TryAdd("data-val-idmobile", ErrorMessage!);
    }
}
```

*`value is null ||` in **Listing 14.13** keeps "required" and "well-formed" as two rules with two messages. `IClientModelValidator` is optional: it only adds `data-val="true"` and `data-val-idmobile` to the `<input>`. Book: Section 14.4.3.*

Model Binding and `ModelState`

Listing 14.14 `OnPost` in `New.cshtml.cs`: an invalid model is shown again with status 422; a valid one is stored and the browser redirected.

```
    public IActionResult OnPost()
    {
        // The attributes have already run during model binding; their verdict is in ModelState.
        if (!ModelState.IsValid)
        {
            Response.StatusCode = 422;          // "understood, but not acceptable"
            return Page();                      // the form comes back with the typed values
        }
        var c = new NewCustomer(Input.Name!.Trim(), Input.Gender!, Input.Birthdate!.Value,
            Input.Place!.Trim(), Input.Address!.Trim(),
            Phone.Format(Phone.Normalize(Input.Phone)!),        // stored as 0812-3456-7890
            Input.Email!.Trim(),
            string.IsNullOrEmpty(Input.Password) ? null : Passwords.Hash(Input.Password));
        string id = store.AddCustomer(c);
        TempData["Flash"] = $"Customer {id} {c.Name} registered.";
        return RedirectToPage("/Bills/Draft", new { customer = id });   // Post/Redirect/Get
```

*Before `OnPost` runs, `[BindProperty]` triggers model binding (text to types) and model validation (every attribute, then `Validate`) into `ModelState`. `ModelState.IsValid` is `Page.IsValid` with the same obligation: you must look at it (**Listing 14.14**). Book: Section 14.4.4.*

Tag Helpers Draw the Form

Listing 14.15 From `New.cshtml`: the summary, the name and the phone, each drawn from the model by tag helpers.

```
<form method="post">
    <div asp-validation-summary="All" class="summary"><strong>Please correct the following:</strong></div>

    <label asp-for="Input.Name">Name</label>
    <input asp-for="Input.Name">
    <span asp-validation-for="Input.Name" class="error"></span>
    <label asp-for="Input.Phone">Mobile phone</label>
    <input asp-for="Input.Phone" placeholder="0812-3456-7890">
    <span asp-validation-for="Input.Phone" class="error"></span>
```

`asp-for` writes `name`, `id`, `value`, `type` and the `data-val-*` attributes; `asp-validation-for` is the place for one message; `asp-validation-summary="All"` is the `ValidationSummary`.

*After a failed POST the form in **Listing 14.15** is drawn again from the bound model: that is how Razor Pages "remembers" what was typed, the job view state did in Web Forms. Book: Section 14.4.5.*

Unobtrusive Validation: Rules as HTML

```
<input placeholder="0812-3456-7890"
       type="text"
       data-val="true"
       data-val-idmobile="Enter an Indonesian mobile number such as 0812-3456-7890 or &#x2B;62 812 3456 7890."
       data-val-required="Enter a mobile phone number."
       id="Input_Phone"
       name="Input.Phone"
       value="">
```

The real HTML of the phone input: `data-val-required` from `[Required]`, `data-val-idmobile` from our attribute. `[Compare]` gives `data-val-equalto`, `[RegularExpression]` gives `data-val-regex`.

The page contains no validation JavaScript of its own: jQuery Validation and Microsoft's unobtrusive adapter read the attributes when the page loads. Nothing was written twice by hand, except the custom rule. Book: Section 14.4.6.

The Browser Half of `[IndonesianMobile]`

Listing 14.16 `indonesian-mobile.js`: a jQuery Validation method called `idmobile` and an adapter that connects it to `data-val-idmobile`.

```
// The browser half of [IndonesianMobile]. The server half is Models/Phone.cs;
// both apply the same rule, and only the server's verdict counts.
(function ($) {
    function normalize(raw) {
        var s = String(raw).replace(/[\s\-.()]/g, "");
        if (s.indexOf("+62") === 0) s = "0" + s.slice(3);
        else if (s.indexOf("62") === 0) s = "0" + s.slice(2);
        return /^08[1-9][0-9]{7,10}$/.test(s) ? s : null;
    }
    $.validator.addMethod("idmobile", function (value, element) {
        return this.optional(element) || normalize(value) !== null;
    });
    $.validator.unobtrusive.adapters.addBool("idmobile");   // reads data-val-idmobile
})(jQuery);
```

*In **Listing 14.16**, `this.optional(element)` is `true` for an empty field, as `IsValid` is for `null`. The PHP, Java and Flask tracks have no client-side validation at all: the server alone is enough for correctness. Book: Section 14.4.6.*

Stopped in the Browser



Figure 14.9 The customer form stopped in the browser: eight messages, each next to its field and all of them in the summary. The browser sent nothing. The date-of-birth rule of `Validate` is missing here, because it exists only on the server.

***Figure 14.9**: the screenshot program counted zero POSTs. The server's seventh message, about a birth date in 2031, is absent here: it lives only in `Validate`. Client validation is a subset of server validation, never the other way round. Book: Section 14.4.6.*

A Line of the Bill Draft

Listing 14.17 `BillLineInput`: a service, whole units from 1 to 999, and a rack from `A01` to `C20`. There is no price.

```
public class BillLineInput
{
    [Required(ErrorMessage = "Choose a service.")]
    public string? ServiceId { get; set; }

    [Required(ErrorMessage = "Enter the number of units.")]
    [Range(1, 999, ErrorMessage = "Enter a whole number from 1 to 999; round kilograms up.")]
    public int? Units { get; set; }

    [Required(ErrorMessage = "Choose a rack.")]
    [RegularExpression("^[A-Ca-c](0[1-9]|1[0-9]|20)$",
        ErrorMessage = "A rack is a letter A to C and a number 01 to 20, for example B07.")]
    public string? Rack { get; set; }
}
```

*`[Range(1, 999)]` on an `int?` in **Listing 14.17** states two business rules: units are whole (3.2 kg is rounded up to 4) and positive. Three more rules need the database and are added in the handler with `ModelState.AddModelError`. Book: Section 14.4.7.*

Worked Example 4: Rules That Need the Server

```
add 0 units, rack D01       422: Enter a whole number from 1 to 999; round kilograms up. | A rack is a letter A to
C and a number 01 to 20, for example B07.
add 2.5 units               422: Enter a whole number from 1 to 999; round kilograms up.
add service S0099           422: Choose a service from the price list.
add rack B08                422: Rack B08 is already taken at O01; choose another.
add rack B09 again          422: Rack B09 is already in this draft.
```

EXAMPLE. A cashier adds 5 kg of wash-and-fold (S0001, Rp 8.000 per kg) on rack `B09` and one bed cover (S0008, Rp 35.000 per item) on rack `b10`. The server looks the prices up: $5 \times 8000 = 40000$ and $1 \times 35000 = 35000$, so the draft total is $40000 + 35000 = 75000$, shown as Rp 75.000. A third line of 2 kg ironing (S0003, Rp 6.000 per kg) adds $2 \times 6000 = 12000$ and the total becomes Rp 87.000; removing it brings the total back to Rp 75.000.

`2.5` fails during binding; `S0099`, rack B08 at O01 and a second B09 are contradictory data: well-formed alone, wrong only in the light of what the server knows. Book: Section 14.4.7.

HTTP Forgets

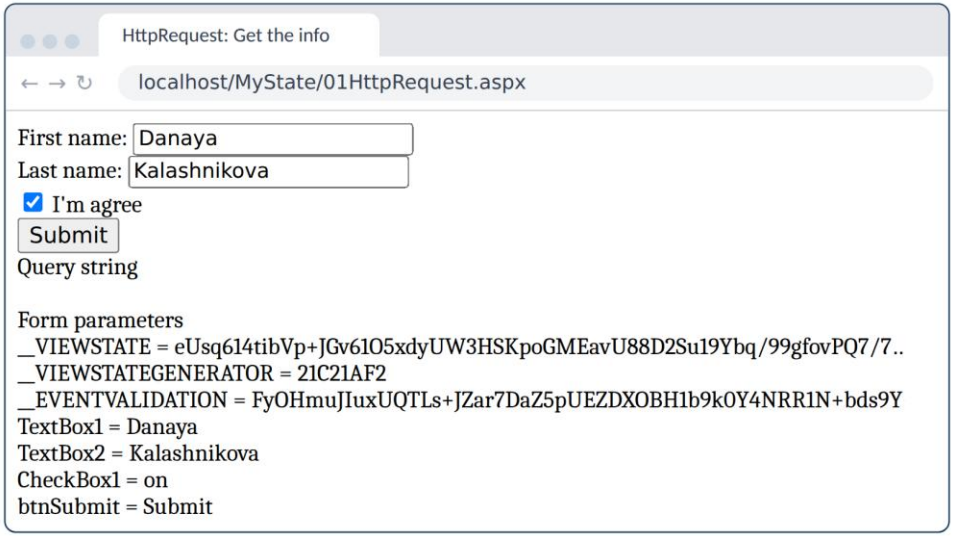


Figure 14.10 The form parameters of one postback: the three fields the person filled in, the button that was pressed, and three hidden fields that Web Forms added, `__VIEWSTATE`, `__VIEWSTATEGENERATOR` and `__EVENTVALIDATION`.

*Every request meets a new instance of the page, thrown away after the answer. **Figure 14.10** is the whole secret of Web Forms' "memory": state written into the last response as hidden fields and sent back. Nothing on the server remembered anything. Book: Section 14.5.1.*

A Map of the Mechanisms

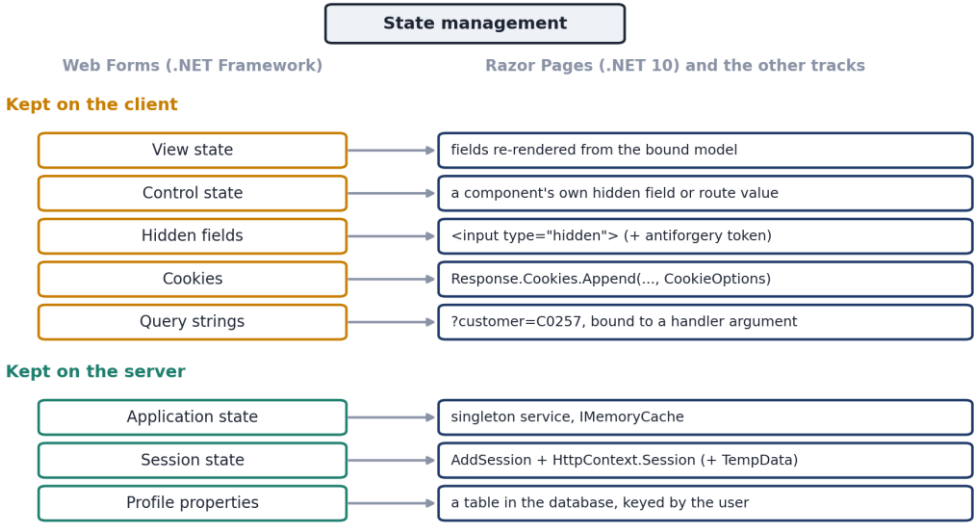


Figure 14.11 The state-management options of ASP.NET Web Forms, grouped by where the data is kept, with the tool that does the same job in Razor Pages today.

***Figure 14.11**: view state, control state, hidden fields, cookies and query strings keep data on the client; application state, session state and profile properties on the server. A redirect with `?name=` printed by `<%= %>` is also an XSS hole: use `<%: %>`. Book: Sections 14.5.2 and 14.5.3.*

Page, Session, Application

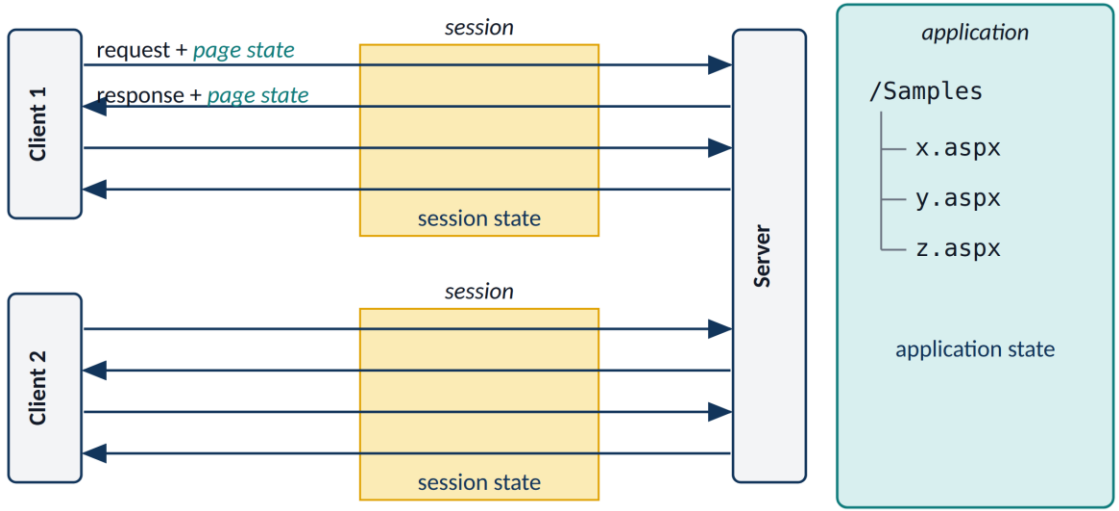


Figure 14.12 Page state travels with each request and response; session state belongs to one client's series of requests; application state belongs to every page of the application and is shared by all clients.

***Figure 14.12**: **view state** lives in one page's hidden field; **session state** is global data for one user, cleared after 20 idle minutes; **application state** is global data for all users, cleared on restart. Book: Section 14.5.3.*

Who Sees It, and for How Long?

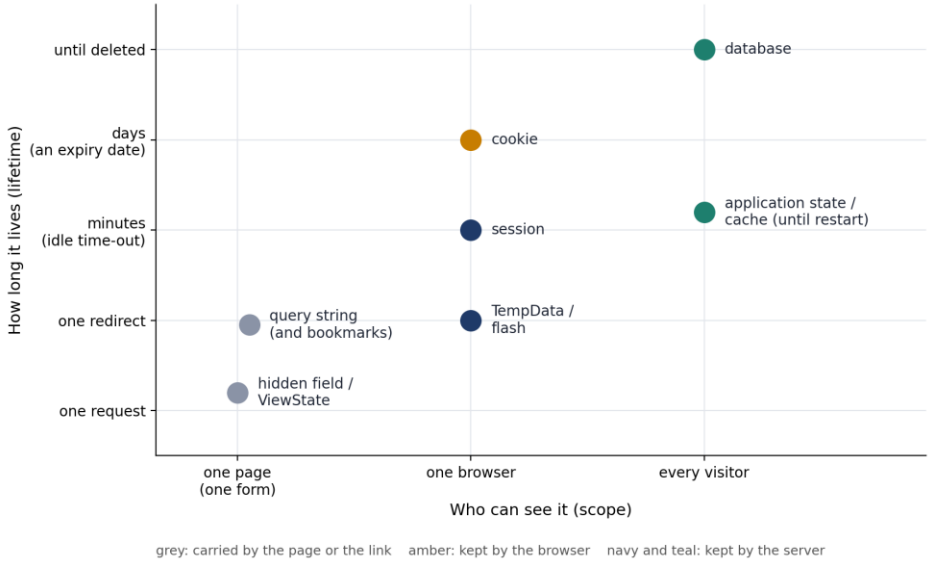


Figure 14.13 Each state mechanism by its scope (who can see it) and its lifetime (how long it lives). Choosing a mechanism means choosing a point on this chart.

*On **Figure 14.13**: the bill draft is one browser for minutes (session); the counter's outlet is one computer for weeks (cookie); "Customer registered" survives one redirect (TempData); the price list is everyone's (cache); the bills are for years (database). Book: Section 14.5.3 and Table 14.4.*

View State, Seen from the Browser

view-source:localhost/MyState/05HiddenField.aspx

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head><title>
    Hidden Field
</title></head>
<body>
    <form method="post" action="./05HiddenField.aspx" id="form1">
<div class="aspNetHidden">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="eov/0rX
 D7PB6hN0Irwk…" />
</div>

<div class="aspNetHidden">
    <input type="hidden" name="__VIEWSTATEGENERATOR" id="__VIEWSTATEGEN
 ERATOR" value="ED5EA427" />
    <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATIO
 N" value="VrW179/mmj8ozmI…" />
</div>
    <div>
            
        <input type="submit" name="Button1" value="No Action Button" id
 ="Button1" /> 
        <input type="hidden" name="HiddenField1" id="HiddenField1" valu
 e="2" />
    </div>
        <strong>         <br />
           Click counter</strong> (Saved in Hidden Field) :&n
 bsp;
        <span id="Label1">2</span>
    </form>
</body>
</html>
```

Figure 14.14 The source of a Web Forms page as the browser received it: `__VIEWSTATE` with a long Base64 value (shortened here, as are the other two, to its first characters and an ellipsis), `__VIEWSTATEGENERATOR`, `__EVENTVALIDATION`, and the page's own hidden field `HiddenField1` with the value `2`.

*In **Figure 14.14**: view state is Base64, **not encrypted** by default but protected by a MAC against tampering; it costs bandwidth twice; it can be switched off per machine, application, page or control. Control state survives when view state is off. Razor Pages simply has none. Book: Sections 14.6.1 and 14.6.2.*

Hidden Fields and Query Strings

A hidden field travels with a POST form, and anyone can edit it. The draft's hidden `customer` is checked against the session:

```
tampered hidden customer    -> /: This draft has expired. Please open it again.
```

A query string is bookmarkable and shareable: `/bills/draft?customer=C0257`, checked against the customer table. There is no price in any form (rule BR-1).

> **Never put a secret, a session id or personal data in a query string.** Addresses are logged by servers and proxies, stored in history, and sent to other sites in the `Referer` header.

Every Razor Pages POST form also carries `__RequestVerificationToken`, the antiforgery token. Book: Sections 14.6.3 and 14.6.4.

The Attributes of a Cookie

Table 14.4 The attributes of a cookie, and the Laundry outlet cookie.

Attribute	Example	What it does	Laundry outlet cookie
`Expires` / `Max-Age`	`Max-Age=2592000`	Keeps the cookie until a date, or for a number of seconds; without either, until the browser closes	30 days
`Path`	`Path=/`	Sends the cookie only for addresses under this path	`/`
`Domain`	`Domain=example.com`	Also sends it to subdomains; omitted, only the exact host	omitted
`Secure`	`Secure`	Sends the cookie only over HTTPS, never over plain HTTP	when the site runs on HTTPS
`HttpOnly`	`HttpOnly`	Hides the cookie from JavaScript (`document.cookie`); only HTTP carries it	yes
`SameSite`	`SameSite=Lax`	`Strict`: never sent with requests started by another site. `Lax`: sent when the person follows a link, but not with a POST, an image or a frame from another site. `None`: always (then `Secure` is required)	`Lax`

***Table 14.4**: `Secure` stops sniffing on open Wi-Fi, `HttpOnly` stops an injected script, `SameSite` stops a cross-site POST carrying your cookies. With `Expires` or `Max-Age` a cookie survives restarts; without, it dies with the browser. Book: Section 14.6.5.*

The Outlet Cookie

Listing 14.18 `Read` and `Write` in `OutletCookie.cs`: 30 days, `HttpOnly`, `SameSite=Lax`, `Secure` on HTTPS; accepted only if it names a real outlet.

```
    public static string? Read(HttpRequest request, ILaundryStore store) =>
        request.Cookies.TryGetValue(Name, out var id) && store.Outlets().Any(o => o.Id == id)
            ? id
            : null;

    public static void Write(HttpResponse response, string outletId, bool https) =>
        response.Cookies.Append(Name, outletId, new CookieOptions
        {
            MaxAge = TimeSpan.FromDays(30),
            HttpOnly = true,                 // page scripts cannot read it
            Secure = https,                  // over HTTPS, never sent over plain HTTP
            SameSite = SameSiteMode.Lax,     // not sent with cross-site POSTs
            IsEssential = true,
            Path = "/",
        });
```

*The outlet is the perfect cookie value: it belongs to the device, lives for weeks, is not secret, and is checked on every read (**Listing 14.18**). The browser received `Set-Cookie: outlet=002; max-age=2592000; path=/; samesite=lax; httponly`. Book: Section 14.6.5.*

How a Session Works

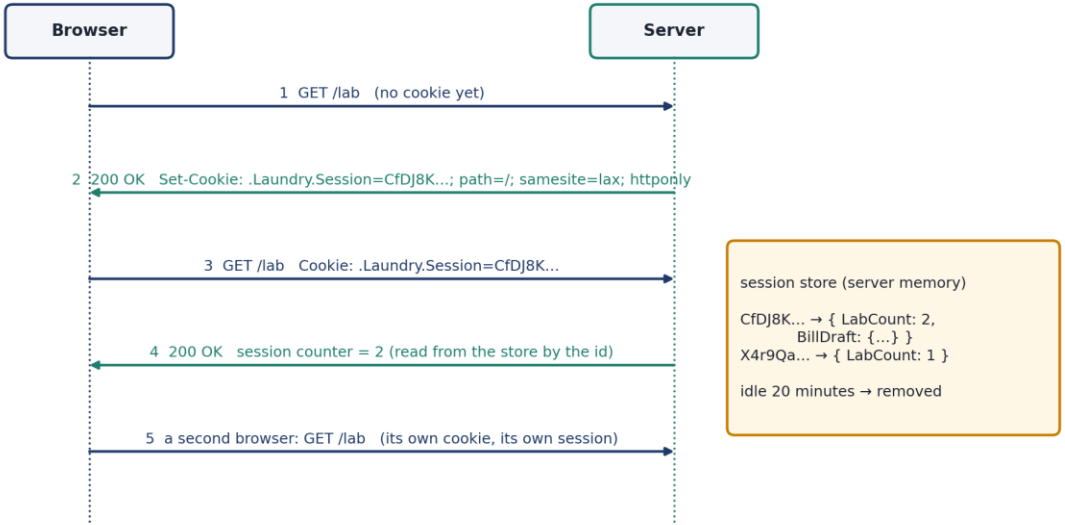


Figure 14.15 How a session works. The first response sets a cookie holding a random id; every later request sends it back; the server uses the id to find that browser's data in its session store. A second browser has its own cookie and therefore its own session.

Figure 14.15*: the browser holds only a random, unguessable key; the server holds the data. A request without a known key gets a new empty session; an idle session is deleted. The session id is a credential. Book: Section 14.7.1.*

Sessions in ASP.NET Core

Listing 14.19 `SessionJson.cs`: `Get<T>` and `Set<T>` store any object in the session as JSON text.

```
using System.Text.Json;

namespace LaundryState.State;

/// <summary>ISession stores bytes and strings only; objects travel as JSON.</summary>
public static class SessionJson
{
    public static T? Get<T>(this ISession session, string key) =>
        session.GetString(key) is { } json ? JsonSerializer.Deserialize<T>(json) : default;

    public static void Set<T>(this ISession session, string key, T value) =>
        session.SetString(key, JsonSerializer.Serialize(value));
}
```

*`AddDistributedMemoryCache()`, `AddSession(…)` and `app.UseSession()` before `MapRazorPages()`. The session stores bytes and strings, so **Listing 14.19** uses JSON, and a changed object must be `Set` again, or the change is lost. Book: Section 14.7.2.*

A Bill Draft in the Session

Listing 14.20 From `OnPost` in `Draft.cshtml.cs`: the hidden customer must agree with the session, and three rules that need the database are added to `ModelState`.

```
    public IActionResult OnPost(string? action, string? customer, int? index)
    {
        var draft = HttpContext.Session.Get<BillDraft>(Key);
        // The hidden field must agree with the session; a stale tab or an expired session does not.
        if (draft is null || draft.CustomerId != customer)
        {
            TempData["FlashError"] = "This draft has expired. Please open it again.";
            return Redirect("/");
        }
                string? outlet = OutletCookie.Read(Request, store);
                string rack = (Line.Rack ?? "").ToUpperInvariant();
                if (prices.Find(Line.ServiceId) is null)
                    ModelState.AddModelError("Line.ServiceId", "Choose a service from the price list.");
                if (ModelState.IsValid && draft.Lines.Any(l => l.Rack == rack))
                    ModelState.AddModelError("Line.Rack", $"Rack {rack} is already in this draft.");
                if (ModelState.IsValid && outlet is not null && store.RackTaken(outlet, rack))
                    ModelState.AddModelError("Line.Rack", $"Rack {rack} is already taken at {outlet}; choose
another.");
```

*The draft works like a shopping cart. In **Listing 14.20** a stale tab, an expired session or an edited field is refused. After a change the handler writes the draft back and redirects; `Confirm` prices every line now from the price list and removes the draft. Book: Section 14.7.3.*

The Draft on Screen

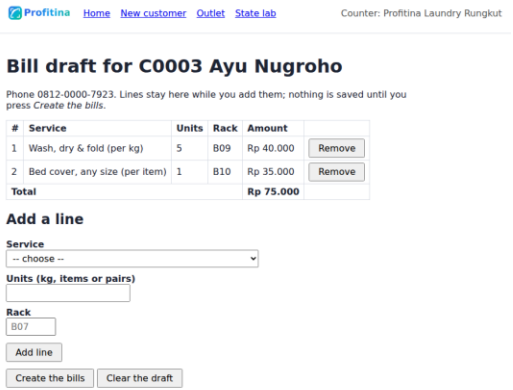


Figure 14.16 The bill draft for customer C0003 at outlet O01 (Rungkut): two lines on racks B09 and B10, their amounts computed by the server, and a total of Rp 75.000.

Figure 14.16*: the amounts are computed by the server; a cashier with developer tools cannot give a friend a discount. Book: Section 14.7.3.*

Post/Redirect/Get with a Flash Message

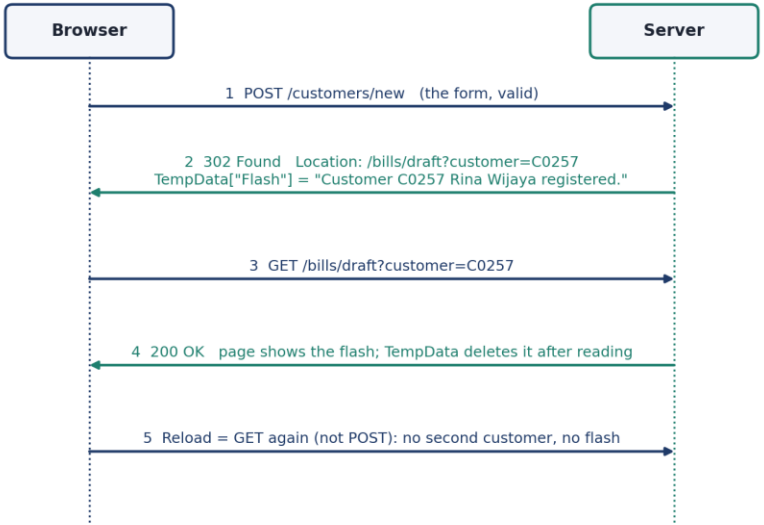


Figure 14.17 Post/Redirect/Get with a flash message. The POST stores a message and answers 302; the GET that follows shows the message and deletes it; a reload repeats only the GET.

Figure 14.17*: a page drawn straight after a POST would let Reload create customer C0258. The GET needs the message, and HTTP forgets:* ***TempData*** *keeps it until it is read once. Book: Section 14.7.4.*

TempData, Checked

```
confirm                  302 -> /
  flash                  Created B2954, B2955 for C0257 Rina Wijaya: Rp 75.000.
  reload                 flash gone
```

Two bills, because the third line was removed; B2954 and B2955 follow the sample data's last bill, B2953; the reload shows nothing.

Here TempData lives in the session (`AddSessionStateTempDataProvider()`); Flask calls it `flash()`, and PHP and Java keep a list in the session.

The layout escapes the message with `@`, and the message holds bill numbers, a name and a total, never a phone or an address. Book: Section 14.7.4.

Application State and the Cache

Listing 14.21 `LabCounter`: application state as a singleton.

```
/// <summary>Application state: one counter shared by every visitor, like Application["Count"].
/// Interlocked makes the increment safe when two requests arrive at the same moment.</summary>
public sealed class LabCounter
{
    private int value;
    public int Next() => Interlocked.Increment(ref value);
}
```

Listing 14.22 `PriceList`: the price list, cached for ten minutes.

```
public sealed class PriceList(ILaundryStore store, IMemoryCache cache)
{
    public IReadOnlyList<Service> All() =>
        cache.GetOrCreate("price-list", entry =>
        {
            entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10);
            return store.LoadServices();
        })!;

    public Service? Find(string? id) => All().FirstOrDefault(s => s.Id == id);
}
```

***Listing 14.21**: `Interlocked` is the modern `Application.Lock()`. **Listing 14.22**: cache public data only. Book: Section 14.7.5.*

Worked Example 5: The State Lab

```
A: GET /lab                app=1 session=1 cookie=1 hidden=0 query=0
A: GET /lab again          app=2 session=2 cookie=2 hidden=0 query=0
A: POST the form           app=3 session=3 cookie=3 hidden=1 query=0
A: POST the form           app=4 session=4 cookie=4 hidden=2 query=0
A: follow the link         app=5 session=5 cookie=5 hidden=0 query=1
A: follow it again         app=6 session=6 cookie=6 hidden=0 query=2
B: GET /lab                app=7 session=1 cookie=1 hidden=0 query=0
```

app grows on every request of every browser; **session** and **cookie** restart at 1 for browser B; **hidden** moves only on POST; **query** only when the link is followed.

After a restart, app and session would start from 1 again, while the cookie counter goes on, because the browser kept it. Exercise 14.8 plays the lab in two browsers. Book: Sections 14.7.6 and 14.7.7.

One Front Desk, Four Tracks

Table 14.5 The four tracks of the chapter, job by job.

Job	Razor Pages	PHP	Java HttpServer	Flask
Declaring rules	Attributes on a model	`if` statements	`if` statements	`if` statements
Browser checks	Unobtrusive, from `data-val-*`	none	none	none
Session store	Distributed memory cache	Files	A `ConcurrentHashMap` written by hand	The signed cookie itself
Session cookie	`.Laundry.Session`	`LAUNDRY_SESSID`	`LAUNDRYSID`	`laundry_session`
Flash messages	`TempData`	A list in `$_SESSION`	A list in the session	`flash()`
Application state	Singleton, `IMemoryCache`	Files under `flock`	`AtomicInteger`, a cached map	Module variables under a `Lock`
CSRF protection	Antiforgery, automatic	Token in session and hidden field	Token in session and hidden field	Token in session and hidden field
Data	In memory from CSV (SQLite optional)	SQLite, PDO	SQLite, JDBC	SQLite, `sqlite3`

*In **Table 14.5**: PHP has no application state (share-nothing) and turns `Input.Name` into `$_POST['Input_Name']`; Java writes its session store by hand with a 256-bit `SecureRandom` id. All four printed the same 42 lines. Book: Sections 14.8–14.11.*

Flask's Session Is the Cookie

```
Set-Cookie: laundry_session=.eJwNzN0Kg...; Expires=Wed, 23 Sep 2026 15:44:38 GMT; HttpOnly; Path=/; SameSite=Lax
Flask session cookie, decoded without any key:
{
  "_permanent": true,
  "csrf": "iWjiP_...",
  "draft": {
    "customer": "C0003",
    "lines": []
  }
}
```

Flask signs the whole `session` with `SECRET_KEY` and sends it as the cookie: tamper-proof, like the view state MAC, but **readable**, decoded here without any key.

Fine for ids and rack numbers; wrong for a phone, a password or a secret, and limited to about 4 KB. `SECRET_KEY` comes from an environment variable, never from the repository. Book: Section 14.10.

Security and Personal Data

The session id is a credential: long and random, only in an `HttpOnly`, `Secure`, `SameSite` cookie, expired when idle, renewed at login.

CSRF: `SameSite=Lax` plus an antiforgery token in session and hidden field; Razor Pages does both automatically.

Never trust client state; escape on output; hash passwords (PBKDF2-HMAC-SHA256, 600,000 iterations), never show or log them.

UU PDP No. 27/2022: purpose, minimization, accuracy, security, limited retention, rights, transparency; choosing a state mechanism is choosing where personal data lives.

Personal data in a session disappears after 20 idle minutes; in a persistent cookie it sits on a shared counter for weeks. Book: Section 14.12 and Table 14.7.

Profitina in Practice

Validation at the edge, always on the server: every scanned address, every post, every community link (the Link Safety screen).

State chosen by scope and lifetime: who is signed in in a session, the weekly score trend in the database, per-tier limits in the application.

Optimistic interfaces, honest servers: a post appears at once, and is restored if the server refuses.

Personal data as a feature: Argon2id hashes, explicit consent, rate-limited logins, export and purge.

The runnable case study remains Profitina Laundry; what carries across is the shape of the discipline. Book: Section 14.13.

Common Mistakes

Validating only in the browser; forgetting `Page.IsValid` or `ModelState.IsValid`; a compare or range validator without `Type`.

A pattern expected to reject empty input; non-nullable input properties; trusting a hidden field, a cookie or a query string.

`<%= Request.QueryString["name"] %>`; showing a page straight after a successful POST; changing a session object without `Set`.

`UseSession()` **after** `MapRazorPages()`; `count++` on shared state; secrets in a Flask session, a cookie or view state.

Cookies without `HttpOnly`, `Secure`, `SameSite`; expecting `Input.Name` in PHP; a session id in the address.

Each mistake comes with its fix in the book. Book: Section 14.14.

Summary: Five Things to Remember

1. Bad data is useless, unauthenticated or contradictory (**Table 14.1**); the browser is a courtesy, the server is the law.

2. Web Forms validators (**Table 14.2**) and Razor Pages data annotations (**Listing 14.10**) express the same rules; `Page.IsValid` and `ModelState.IsValid` must be checked.

3. A custom attribute runs on both sides (**Listing 14.13**, **Listing 14.16**); rules that need the database live in the handler.

4. HTTP forgets; choose a mechanism by scope and lifetime (**Figure 14.13**); cookies need `HttpOnly`, `Secure`, `SameSite` (**Table 14.4**).

5. Sessions keep data on the server behind a random id (**Figure 14.15**); Post/Redirect/Get and TempData make reloads harmless (**Figure 14.17**).

Book: Section 14.15; Exercises 14.1–14.12 with solutions in Sections 14.16 and 14.17.

Next Time

Chapter 15 — ADO.NET and the Data Layer. How a .NET program talks to a database, from the first connection string to the last `Commit()`.

Connected (connection, command, parameters, DataReader) and disconnected (DataAdapter, DataSet, DataTable) work; transactions; a repository in five tracks.

Before then: work Exercises 14.1, 14.3, 14.8 and 14.11.

Today's `SqliteLaundryStore` waited for a NuGet package; Chapter 15 is about exactly that layer. Book: Chapter 15.

Lecture 15 — ADO.NET and the Data Layer

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 15

Lecturer: Irfan Subakti — yifana@gmail.com

Today we open the box every earlier page took for granted: where the data comes from, and how several writes become one act. Book: Chapter 15, Sections 15.1–15.20.

What This Meeting Covers

1. **What ADO.NET is** — one interface, a provider per database, and two ways of working: connected and disconnected.
2. **The connected model** — connection, command, parameters, DataReader; open late, close always.
3. **Transactions** — a bill, its status log and its first payment as one act; isolation levels.
4. **The disconnected model** — the DataSet, row states and versions, DataView, the DataAdapter.
5. **Stored procedures and SQL injection** — one place for a unit of work; gluing shown and prevented.
6. **A data layer** — one repository interface, implemented in C#, PHP, Java and Python; EF Core.

Three questions organize the chapter: what ADO.NET is and why it has two halves; how several writes become one act; where the SQL should live and who may put text into it. Book: Section 15.1.

Where We Left Off

Chapter 5 taught SQL and PDO; Chapter 7 JDBC and a Data Access Object; Chapters 13 and 14 showed data on a page and kept state between requests. In every one of them the pages simply *had* the data.

What ran here: everything in `System.Data` ran on the .NET 10 SDK; the SQLite, MySQL and SQL Server providers and EF Core are NuGet packages and did not. The PHP, Java and Python layers ran against the real Laundry database.

> **KEY INSIGHT.** A page should never open a database connection. It should ask a data layer a question in the language of the shop — “create this bill”, “what is owed on B2954?” — and receive an answer or a sentence explaining why not.

Everything about connections, commands, readers, DataSets and transactions lives inside that layer, which is why it can change without touching a page. Book: Sections 15.1 and 15.1.1.

Universal Data Access

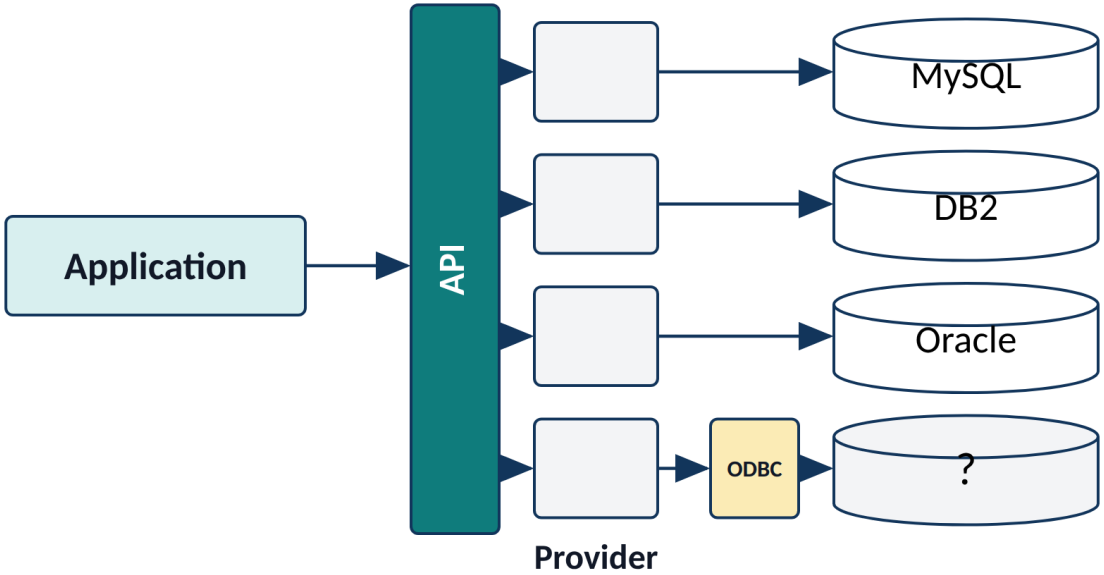


Figure 15.1 Universal data access: the application speaks to one API, behind which a provider for each database (MySQL, DB2, Oracle) translates and an ODBC bridge reaches a source with no provider of its own. The source no longer matters to the application.

*In **Figure 15.1** the application talks only to the API; each provider translates for one database; a bridge such as ODBC reaches sources with no provider of their own. Book: Section 15.2.1.*

What ADO.NET Is

> **DEFINITION. ADO.NET** is the set of .NET classes, in the namespace `System.Data` and its children, through which a .NET program reaches structured data: relational databases above all, but also XML documents and any other source for which a provider exists. It offers a uniform object-oriented interface — connection, command, parameter, reader, transaction, adapter, DataSet — and a **data provider** for each kind of source implements that interface.

Ancestors: ODBC (1992, a C interface), OLE DB (1996, COM), ADO (1996, the Recordset). ADO.NET (2002) is a redesign, not a new ADO: it added the connectionless DataSet and XML.

A connection string that begins `provider=SQLOLEDB` is an OLE DB string: you will still meet all three ancestors in old code. Book: Sections 15.2.1–15.2.2 and Table 15.1.

Two Halves

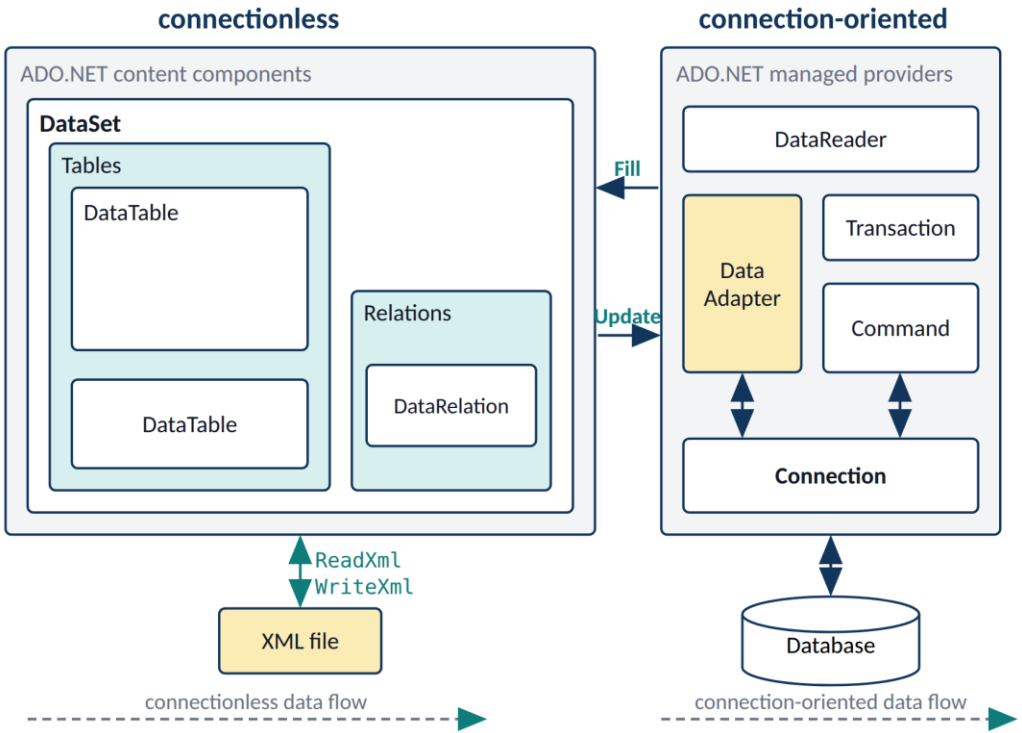

connectionless
connection-oriented
ADO.NET content components
DataSet
Tables
DataTable
DataTable
Relations
DataRelation
Fill
Update
ADO.NET managed providers
DataReader
Data Adapter
Transaction
Command
Connection
ReadXml
WriteXml
XML file
Database
connectionless data flow
connection-oriented data flow

Figure 15.2 ADO.NET's two halves: on the right the **managed providers** (Connection, Command, Transaction, DataReader, DataAdapter) work against the database; on the left the **content components** (a DataSet of DataTables and DataRelations) hold data in memory and read and write XML. The DataAdapter's `Fill` and `Update` are the only bridge between them.

Figure 15.2 *is the picture to keep in mind for the rest of today: the managed providers on one side, the DataSet on the other, and the DataAdapter's* `Fill` *and* `Update` *as the only bridge. Book: Section 15.2.3.*

Connected and Connectionless

DEFINITION. In the **connection-oriented** model the program keeps a connection open while it works and reads the result row by row through a DataReader. In the **connectionless** model a DataAdapter opens a connection briefly to *fill* a DataSet in main memory, the program works on the DataSet as long as it likes, and the adapter connects briefly again to *update* the source.

The time each model holds a connection is Equation (15.1):

$$T_{\text{connected}} = t_{\text{query}} + t_{\text{use}}, \qquad T_{\text{disconnected}} = t_{\text{fill}} + t_{\text{update}} \tag{15.1}$$

A connection is scarce and pooled, so the model that holds each one for less time serves more visitors. In a web page t_{use} is tiny if rows are read into a list first. Book: Section 15.2.3.

Which Model for Which Job

Table 15.1 The connected and disconnected models. A web request is short, so web pages mostly use the connected model — open late, close early — and use in-memory tables for work that happens after the rows have arrived.

	Connection-oriented	Connectionless
Connection	kept open while the program works	opened only for `Fill` and `Update`
Where the data is	in the database; rows stream past	copied into main memory (a DataSet)
Changes	sent at once, statement by statement	kept in memory, then written back together
Suits	short transactions, few parallel users, data that must be up to date	many parallel and long-lasting accesses, work while offline
Risk	holding connections too long	conflicting changes: someone else changed the row meanwhile
Laundry example	the cashier's "create bill" button	a report built once and sorted twenty ways

***Table 15.1**: read once from first row to last with a DataReader; use a DataSet to cache, relate, bind or process data without holding a connection. A DataAdapter itself uses a DataReader inside `Fill`. Book: Sections 15.2.3 and 15.2.4.*

How Long a Connection Is Held

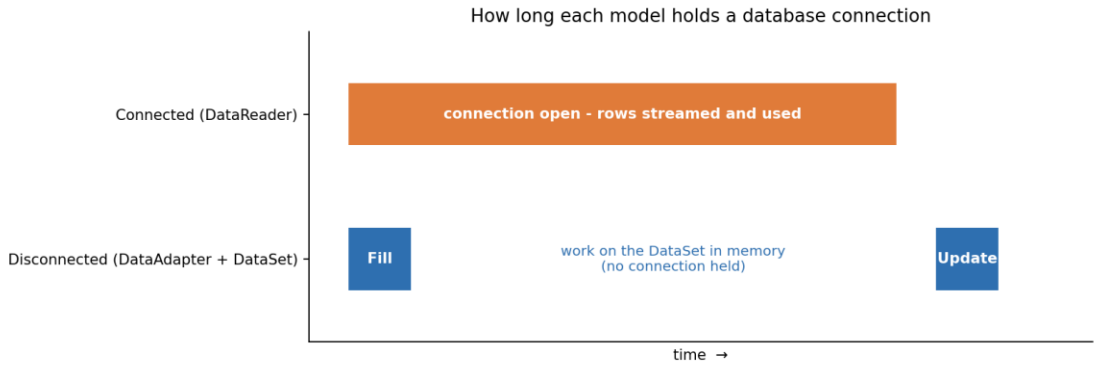


Figure 15.3 How long each model holds a database connection. The connected model holds one for as long as rows are being used; the disconnected model holds one only while filling and while writing back.

***Figure 15.3**: a reader holds its connection from the first row to the last, including any rendering done in the same loop; an adapter holds one only during `Fill` and `Update`. Book: Section 15.2.3.*

Providers Are Packages Now

```
dotnet add package Microsoft.Data.Sqlite          # SQLite: the Laundry database file
dotnet add package MySqlConnector                  # MySQL 8 or MariaDB (XAMPP)
dotnet add package Microsoft.Data.SqlClient        # SQL Server, SQL Server Express, Azure SQL
dotnet add package Microsoft.EntityFrameworkCore.Sqlite   # EF Core on SQLite (Section 15.16)
```

`System.Data` and `System.Data.Common` stayed in the framework; `System.Data.SqlClient` became `Microsoft.Data.SqlClient`; SQLite, MySQL and PostgreSQL providers are NuGet packages.

> **WARNING:** `sa` (SQL Server) and `root` (MySQL) can do anything to every database. An application must connect with a login that can reach one database and do only what the application does.

SQL Server Developer and Express are free; SSMS manages it on Windows; Northwind is its classic sample. The Laundry code needs no server: SQLite and CSV copies of the database. Book: Sections 15.2.5, 15.3 and Table 15.3.

The Connected Model

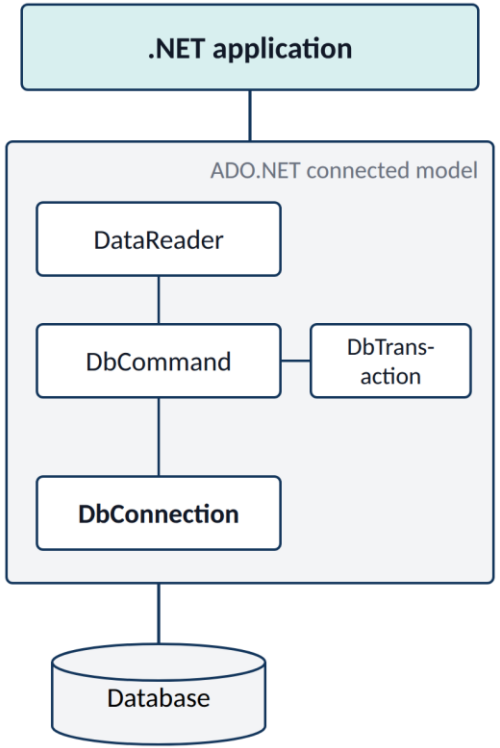


Figure 15.4 The ADO.NET connected model. A **DbConnection** represents the connection to the data source; a **DbCommand** represents one SQL statement; a **DbTransaction** groups commands so that they succeed or fail together; a **DataReader** delivers the result of a query one row at a time.

***Figure 15.4**: `DbConnection`, `DbCommand`, `DbTransaction`, and a DataReader. Each is an interface (`IDbConnection`, …), a class per provider (`SqlConnection`, …), and since .NET 2.0 an abstract base class in `System.Data.Common`. Book: Section 15.4.1.*

Interfaces and Providers

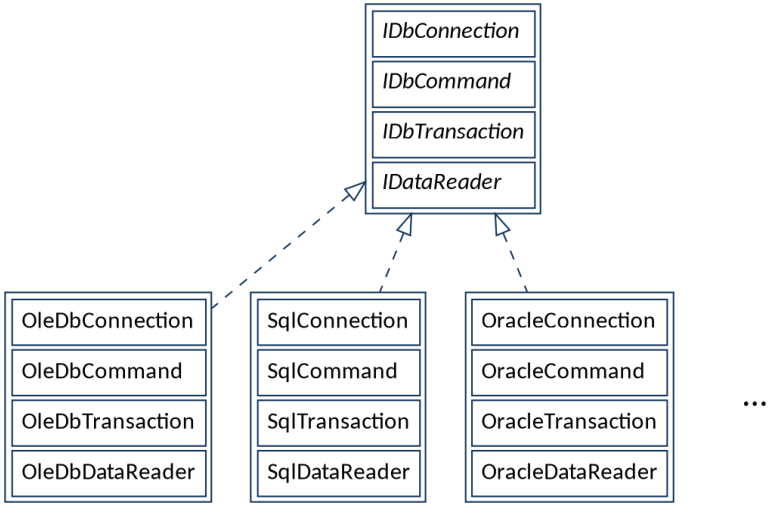


Figure 15.5 The class hierarchy. Four interfaces at the top; below them, one family of four classes per provider (OLE DB, SQL Server, Oracle, …), each class implementing its interface.

***Figure 15.5**: programming to the interface, or to the Db… base class, is what makes ADO.NET "universal"; only the line that creates the connection names a provider. Book: Section 15.4.1.*

The Pattern of Connected Access

Pseudocode 15.1 The pattern of connection-oriented data access. Lines 9–13 run on success and on failure alike, so a connection is never left open.

```
CONNECTED-ACCESS()
1  declare the connection, with no value yet
2  try
3      request a connection to the database and open it
4      execute an SQL statement
5      process the result
6      release the resources used by the result (the reader)
7  catch an exception
8      handle it: log it, and tell the user one sentence
9  finally
10     try
11         if the connection was created, close it
12     catch an exception
13         handle it
```

Pseudocode 15.1*: a connection that is not closed is not returned to the pool; a page that leaks one per error will, on a bad day, exhaust the pool and stop the whole site. Book: Section 15.4.2.*

EmployeesReader, as It Was Written

Listing 15.1 The body of the `try` in `EmployeesReader.OleDb.cs`: create, open, command, reader, loop. Not run in this book (Windows, SQL Server and OLE DB).

```
            con = new OleDbConnection(connStr);
            con.Open(); // Open connection
            // 2. Execute command
            // --- Create SQL command
            IDbCommand cmd = con.CreateCommand();
            cmd.CommandText = "SELECT EmployeeID, LastName, " +
                "FirstName FROM Employees";
            // --- Execute SQL command; result is an OleDbDataReader
            IDataReader reader = cmd.ExecuteReader();
            object[] dataRow = new object[reader.FieldCount];
            // 3. Process result
            while (reader.Read()) {
                int cols = reader.GetValues(dataRow);
                for (int i = 0; i < cols; i++) {
                    Console.Write("| {0} ", dataRow[i]);
                }
                Console.WriteLine();
            }
```

*In **Listing 15.1** every variable has an interface type except `new OleDbConnection(connStr)`; `CreateCommand()` returns a command of the connection's own provider; `GetValues` fills an array sized by `FieldCount`. The `finally` closes the connection. Book: Section 15.4.3 and Figure 15.14.*

EmployeesReader, as It Is Written Today

Listing 15.2 `EmployeesReader.SqlClient.cs`: .NET 10, `await using` closing reader, command and connection, and the connection string from the environment. Not run in this book.

```
using Microsoft.Data.SqlClient;

string connStr = Environment.GetEnvironmentVariable("NORTHWIND")
    ?? throw new InvalidOperationException("Set NORTHWIND to a connection string.");

await using var con = new SqlConnection(connStr);          // closed automatically, even on error
await con.OpenAsync();
await using var cmd = new SqlCommand("SELECT EmployeeID, LastName, FirstName FROM Employees", con);
await using SqlDataReader reader = await cmd.ExecuteReaderAsync();
while (await reader.ReadAsync())
{
    Console.WriteLine($"| {reader.GetInt32(0)} | {reader.GetString(1)} | {reader.GetString(2)}");
}
```

***Listing 15.2**: `await using` does the work of the `finally` block; the `…Async` calls free the thread while the database works; the password lives in the environment or `dotnet user-secrets`, never in source. Book: Section 15.4.3.*

Connection Strings

Table 15.2 Connection strings for the providers this chapter uses. The keys differ from provider to provider; each provider's documentation lists them, and each has a `…ConnectionStringBuilder` class that writes them for you without typing errors.

Provider	Example connection string
`Microsoft.Data.Sqlite`	`Data Source=data/laundry.db;Mode=ReadWrite;Foreign Keys=True;Default Timeout=5`
`MySqlConnector`	`Server=localhost;Port=3306;Database=profitina_laundry;User ID=laundry;Password=…`
`Microsoft.Data.SqlClient`	`Server=localhost;Database=Northwind;Integrated Security=true;TrustServerCertificate=true`
`Microsoft.Data.SqlClient` (SQL login)	`Server=db.example.com;Database=Laundry;User ID=laundry_app;Password=…;Encrypt=true`

Never a password in source code; **build strings with the builder class**, not concatenation; **leave pooling on**, so "open late, close early" costs microseconds.

***Table 15.2**: `key=value` pairs separated by semicolons; the keys differ between providers. Book: Section 15.4.4.*

Three Ways to Execute a Command

Table 15.3 The three ways to execute a command. The number returned by `ExecuteNonQuery` is worth checking: an `UPDATE … WHERE id = @id AND status = 'queued'` that returns 0 means somebody else changed the row first.

Method	Returns	Use it for	In the Laundry data layer
`ExecuteReader()`	a DataReader over the rows	`SELECT` returning rows	`RecentBills`, `FindBill`, `SearchCustomers`
`ExecuteNonQuery()`	`int`: rows affected	`INSERT`, `UPDATE`, `DELETE`, DDL	the three `INSERT`s; `UPDATE Bill SET status…`
`ExecuteScalar()`	`object`: first column of first row, or `null`	one value: `COUNT(*)`, `MAX(id)`, a price	`SELECT price…`, `SELECT MAX(id) FROM Bill`

*In **Table 15.3** the number returned by `ExecuteNonQuery` is worth checking: 0 from `UPDATE … WHERE id = @id AND status = 'queued'` means somebody changed the row first. Book: Section 15.4.5.*

Parameters

> **DEFINITION.** A **parameterized command** (a *prepared statement* in PHP and Java) is a command whose SQL text is fixed and contains placeholders, and whose values are sent to the database separately from that text. The database parses the text first, decides what the statement means, and only then receives the values — as data, which can never change what the statement means.

A parameter has a **name**, a **value**, a **type** (`DbType`) and a **direction**: `Input`, `Output`, `InputOutput`, `ReturnValue`.

Placeholders: `@name` for SqlClient, Sqlite and MySqlConnector; `?` by position for OLE DB, ODBC and JDBC; `?` or `:name` in PDO and Python.

Three steps: placeholders in the SQL, a parameter object per placeholder, values assigned before execution. Book: Section 15.4.6 and Table 15.6.

One Helper for Every Command

Listing 15.3 `AdoBillRepository.Command`: every value becomes a `DbParameter`, and `null` becomes `DBNull.Value`, the database's NULL.

```
    /// <summary>A command with its text and its parameters; values never touch the text.</summary>
    private static DbCommand Command(DbConnection con, DbTransaction? tx, string sql,
                                     params (string Name, object? Value)[] parameters)
    {
        DbCommand cmd = con.CreateCommand();
        cmd.CommandText = sql;
        cmd.Transaction = tx;
        foreach (var (name, value) in parameters)
        {
            DbParameter p = cmd.CreateParameter();
            p.ParameterName = name;
            p.Value = value ?? DBNull.Value;
            cmd.Parameters.Add(p);
        }
        return cmd;
    }
```

Listing 15.3 *names no provider:* `CreateCommand()` *and* `CreateParameter()` *return the connection's own kind. C#'s* `null` *means "no object"; a database NULL is a value of its own,* `DBNull.Value`*.* `Scalar` *and* `Execute` *wrap the other two execute methods. Book: Section 15.4.6.*

The DataReader

- **ExecuteReader()** returns a DataReader object

```
IDataReader ExecuteReader()
IDataReader ExecuteReader(CommandBehavior behavior);
```

- A DataReader reads the result sequentially, row by row

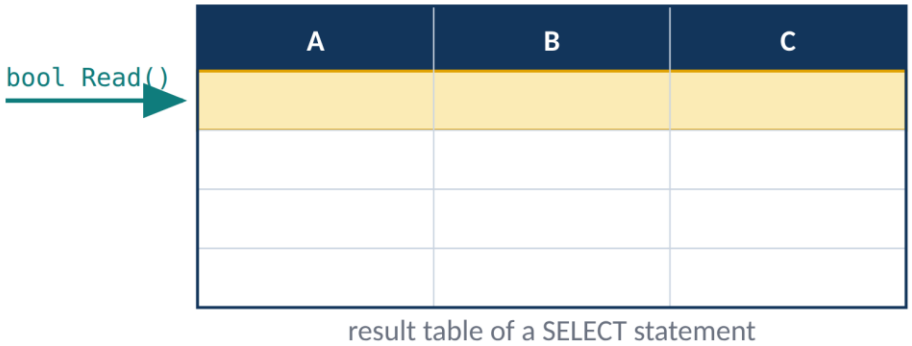


Figure 15.6 A DataReader over the result table of a `SELECT`. `Read()` advances the cursor one row and reports whether there was a row to advance to.

***Figure 15.6**: a cursor that starts before the first row; `Read()` moves on and reports whether there was a row. It cannot go back or jump. Indexers return `object`; typed getters convert; `IsDBNull` asks first. Book: Section 15.4.7.*

A DataReader, Run for Real

Listing 15.4 The loop of `AdoBasics/ReaderDemo.cs` over the Laundry employees, with a `DataTableReader`, the `IDataReader` that ships with `System.Data`.

```
            Console.WriteLine($"FieldCount = {reader.FieldCount}; column 1 is \"{reader.GetName(1)}\"
({reader.GetDataTypeName(1)})");
            int spv = reader.GetOrdinal("spv_id");              // look the position up once
            object[] dataRow = new object[reader.FieldCount];
            while (reader.Read())                               // process: one row at a time
            {
                if ((string)reader["section"] != "transaction") continue;
                int cols = reader.GetValues(dataRow);           // all columns into an array
                for (int i = 0; i < 3; i++)
                    Console.Write("| {0} ", dataRow[i]);
                Console.WriteLine("| supervisor {0}", reader.IsDBNull(spv) ? "(none: the owner)" :
reader.GetString(spv));
            }
```

*In **Listing 15.4**, `GetOrdinal` runs once before the loop, and `IsDBNull` guards `GetString`: the owner, E0001, has no supervisor. `DataTable.Load(reader)` works with every provider. Book: Section 15.4.7.*

Worked Example 1: Three Rows, One Act

Five kilograms of wash-dry-fold at Rp 8.000 per kg; Rp 20.000 paid now. The server computes the charge, Equation (15.2), and the balance, Equation (15.3):

$$\text{charge} = \text{price} \times \text{unit} = 8000 \times 5 = 40000 \tag{15.2}$$

$$\text{balance} = \text{charge} - \sum_{p \in \text{payments of the bill}} \text{amount}_p = 40000 - 20000 = 20000 \tag{15.3}$$

> **DEFINITION.** A **transaction** is a group of statements that the database treats as one indivisible act, with the four **ACID** properties: atomicity, consistency, isolation, durability. It ends with **commit** or **rollback**.

Write the `Bill` and the `StatusLog`, then fail on the `Payment`, and the books say Rp 40.000 is owed while the drawer holds Rp 20.000 more than they explain. Book: Section 15.5.1.

The Unit of Work: Rules First

Listing 15.5 The start of `AdoBillRepository.CreateBillWithFirstPayment`: one connection, one transaction, the price read and the rules checked inside it.

```
    public string CreateBillWithFirstPayment(NewBill b)
    {
        using DbConnection con = Open();
        using DbTransaction tx = con.BeginTransaction();
        try
        {
            object? price = Scalar(con, tx, "SELECT price FROM Service WHERE id = @id", ("@id", b.ServiceId));
            if (price == null)
                throw new DataLayerException($"There is no service {b.ServiceId}.");
            if (b.Unit <= 0)
                throw new DataLayerException("The number of units must be at least 1.");
            decimal charge = Convert.ToDecimal(price) * b.Unit;
            if (b.PayAmount < 0 || b.PayAmount > charge)
                throw new DataLayerException($"A first payment must be between Rp 0 and the charge,
{Money.Rp(charge)}.");
```

*In **Listing 15.5** the charge comes from the price list by Equation (15.2), never from the form. Disposing a transaction that was neither committed nor rolled back **rolls it back**. Book: Section 15.5.2.*

The Unit of Work: Write, Then Commit

Listing 15.6 The rest of `CreateBillWithFirstPayment`: the next id read inside the transaction, the log and the payment written, then `Commit`, or `Rollback` and one sentence.

```
            // The next id is read INSIDE the transaction, so no other cashier can take it.
            object? max = Scalar(con, tx, "SELECT MAX(id) FROM Bill");
            string id = "B" + ((max is string s ? int.Parse(s[1..]) : 0) + 1).ToString("D4");
            InsertLog(con, tx, id, "queued", b.Now, b.EmployeeId);
            if (b.PayAmount > 0)
                InsertPayment(con, tx, id, b.PayMethod ?? "", b.PayAmount, b.PayReference, b.Now);
            tx.Commit();                                    // all three rows, at once
            return id;
        catch (DbException e)
        {
            tx.Rollback();                                  // none of them
            throw new DataLayerException(e.Message, e);
        }
```

*Every command in **Listing 15.6** receives `tx`; the `INSERT` of the bill is elided here. `SELECT MAX(id)` returns `B2953`, so the new bill is `B2954`. A `DbException` becomes a `DataLayerException`, so the page never sees provider types. Book: Section 15.5.2.*

Everything, or Nothing

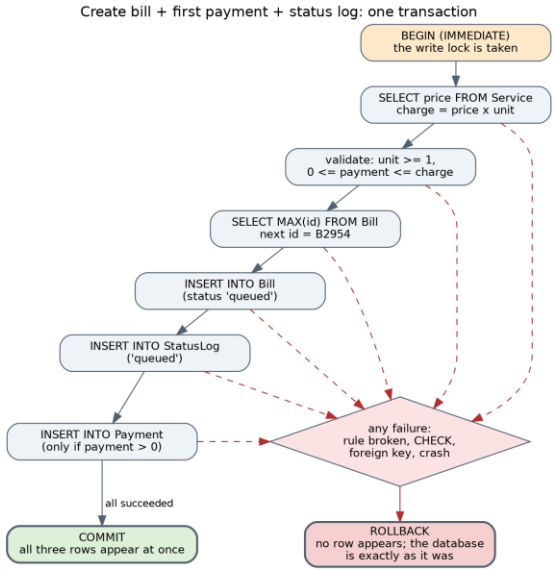


Figure 15.7 Creating a bill with its first payment and status log as one transaction. Any failure after `BEGIN` — a broken rule, a CHECK or foreign-key constraint, a crash — leads to `ROLLBACK`, and the database is exactly as it was; only the path through every step reaches `COMMIT`.

***Figure 15.7**: every failure after `BEGIN` leads to `ROLLBACK`; only the path through every step reaches `COMMIT`. `TransactionScope` and EF Core's `SaveChanges()` are two other ways to spell the same thing. Book: Sections 15.5.2 and 15.5.4.*

A Unit of Work, in General

Pseudocode 15.2 A unit of work. Everything `work` writes becomes visible at line 5, or never.

```
UNIT-OF-WORK(work)
1  open a connection
2  BEGIN a transaction on it
3  try
4      result = work(connection, transaction)    // reads, checks, writes
5      COMMIT
6      return result
7  catch any failure
8      ROLLBACK
9      raise one error the caller understands
10 finally
11     close the connection
```

Pseudocode 15.2 *is implemented five times in the chapter: over ADO.NET, over a DataSet, and in PHP, Java and Python. Book: Section 15.5.2.*

Isolation Levels

- Defines how read and write locks are used in a transaction
- ADO.NET transactions allow different isolation levels

```
public enum IsolationLevel {
    ReadUncommitted, ReadCommitted, RepeatableRead, Serializable, ...
}
```

IsolationLevel	Meaning
ReadUncommitted	reading data locked by another transaction is allowed; *dirty reads* possible
ReadCommitted (default)	reading locked data is prohibited; no dirty reads, but *phantom rows* and *non-repeatable reads* can occur
RepeatableRead	as `ReadCommitted`, and reads are repeatable
Serializable	serialized access; *phantom rows* cannot occur

Figure 15.8 Isolation levels: the usage of read and write locks in a transaction. `ReadUncommitted` allows dirty reads; `ReadCommitted`, the usual default, prevents them but allows phantom rows and non-repeatable reads; `RepeatableRead` makes reads repeatable; `Serializable` serializes access and prevents phantoms.

***Figure 15.8**: isolation is bought with locks, and locks cost concurrency. `ReadCommitted` is the usual default. Book: Section 15.5.3.*

Three Anomalies at the Laundry

Table 15.4 The three anomalies and the lowest isolation level that prevents each. `Snapshot`, a level added later for SQL Server, gives every transaction a consistent picture as of its start without blocking readers.

Problem	What happens	Laundry example	Prevented from
Dirty read	you read a row another transaction has written but not committed	the dashboard counts bill B2954 that the cashier's transaction is about to roll back	`ReadCommitted`
Non-repeatable read	you read a row twice and get two different values, because another transaction committed a change between	a receipt reads the balance, a second cashier records a payment, the receipt's total no longer adds up	`RepeatableRead`
Phantom row	you run the same query twice and a new row appears, because another transaction inserted it	two cashiers both read `MAX(id) = B2953` and both compute `B2954`	`Serializable`

*The phantom of **Table 15.4** is the next-id problem. SQLite has one writer at a time: `BEGIN IMMEDIATE` takes the write lock at once. MySQL's InnoDB needs `FOR UPDATE`. An auto-increment key avoids the question. Book: Section 15.5.3.*

Worked Example 2: Two Cashiers, Two Connections

Listing 15.7 The real output of `python/demo_isolation.py`: what connection B sees while connection A is inside a transaction.

```
A inside its transaction sees 2954 bills
B at the same moment sees     2953 bills  (A's rows are not committed)
B tries to start writing:      OperationalError: database is locked
after A commits, B sees       2954 bills
A deletes B2954 but has not committed; B sees 2954
A rolls back; B sees          2954 bills  (the delete never happened)
```

Step 1. A inserts B2954 inside a transaction: A sees 2954 bills, B still 2953.

Step 2. B cannot begin writing while A holds the lock; after five seconds, `database is locked`.

Step 3. After `COMMIT` B sees 2954; A's uncommitted delete is never visible, and `ROLLBACK` undoes it.

Listing 15.7 *is isolation and atomicity watched from outside. Exercise 15.6 traces the same two cashiers step by step. Book: Section 15.5.3.*

The Disconnected Model

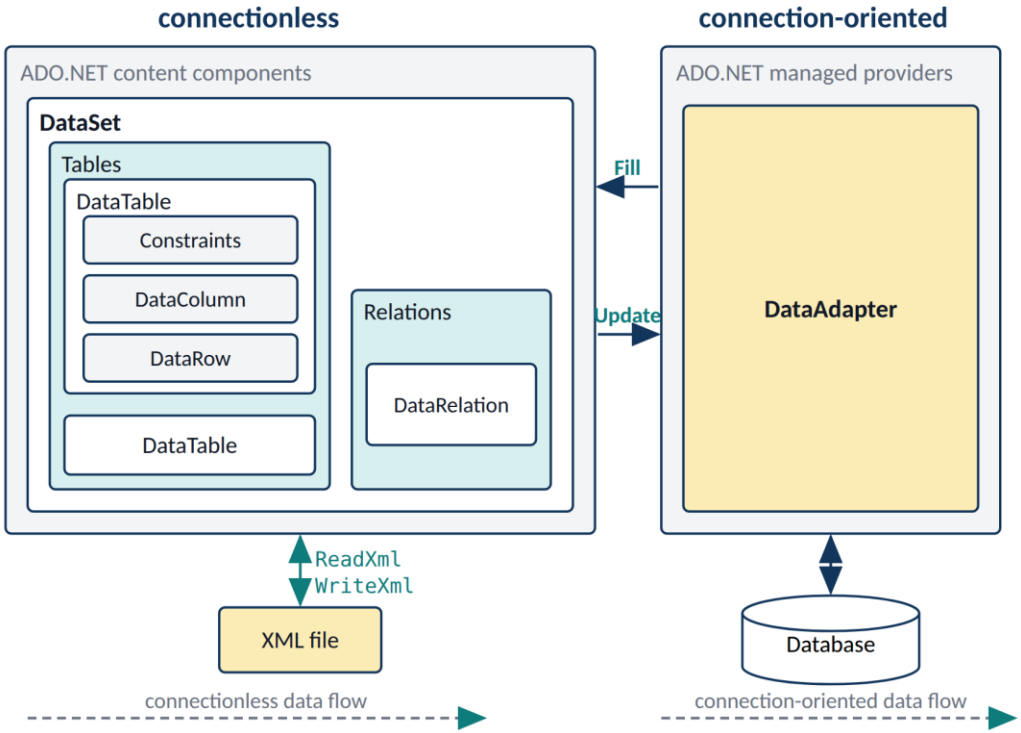


Figure 15.9 The connectionless architecture: inside the DataSet, DataTables with Constraints, DataColumns and DataRows, and DataRelations between tables. The DataAdapter fills it and updates the database from it; `ReadXml` and `WriteXml` connect it to XML files.

*Many parallel, long-lasting accesses would need a connection each. **Figure 15.9**: cache the data in a main-memory database and use only short connections to fill and to update. The price is that conflicting changes become possible. Book: Section 15.6.1.*

The Structure of a DataSet

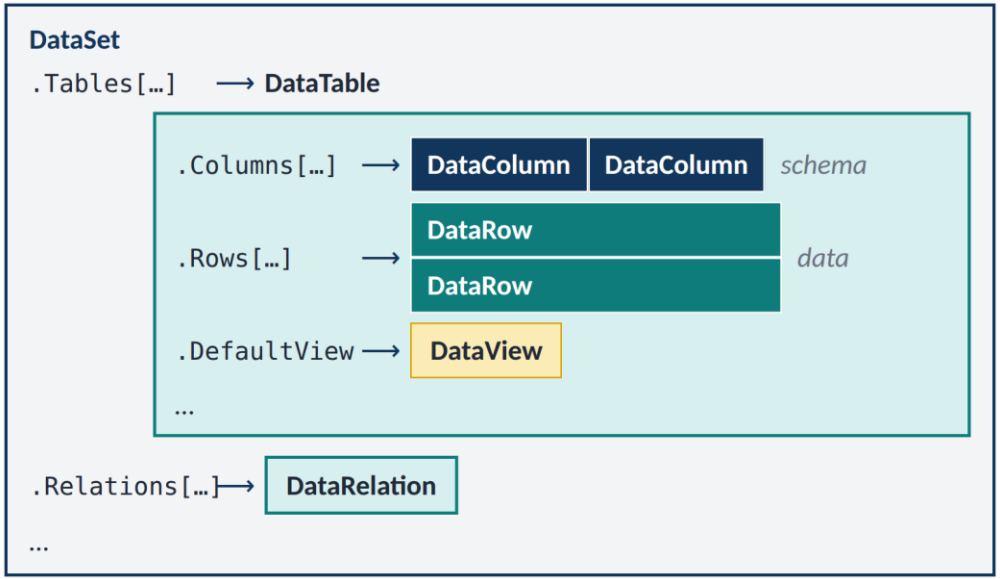
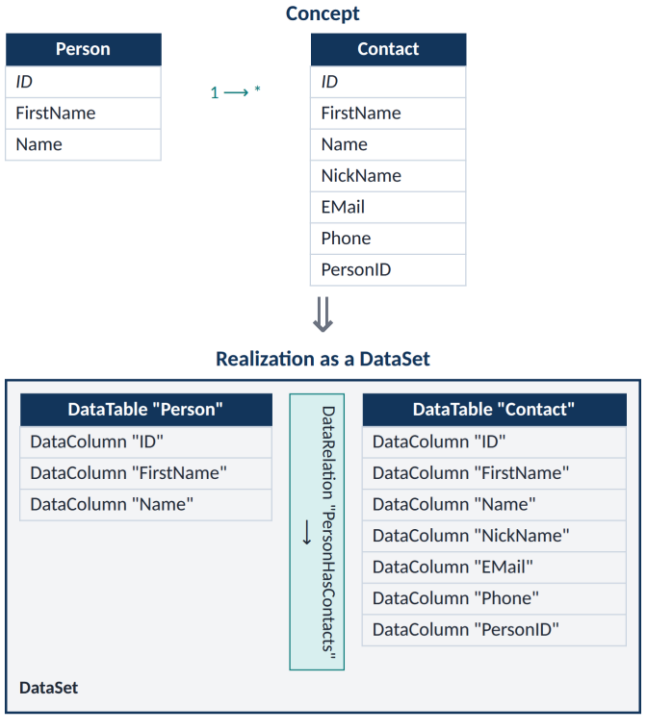


Figure 15.10 The structure of a DataSet. `.Tables[…]` reaches a DataTable; `.Columns[…]` its DataColumns (the schema); `.Rows[…]` its DataRows (the data); `.DefaultView` its DataView; `.Relations[…]` the DataRelations between tables.

***Figure 15.10**: each dot in `ds.Tables["Bill"].Rows[0]["status"]` steps one box inward. Columns are the schema, rows the data; `PrimaryKey` is an array of columns; a DataRelation joins a parent and a child. Book: Section 15.6.2.*

Persons and Their Contacts



Implementation steps: define the schema; define the data; access the data.

Figure 15.11 The person-contacts example: in concept a person has any number of contacts. It is realized as two DataTables, `Person` and `Contact`, joined by the DataRelation `PersonHasContacts`.

Figure 15.11*: a DataSet built with no database at all, in three steps: define the schema, define the data, access the data. Book: Section 15.6.3.*

A Relation and Two Rows

Listing 15.8 From `AdoBasics/MyAdo.cs`: the relation `PersonHasContacts`, one person and one contact.

```
        DataColumn parentCol = ds.Tables["Person"]!.Columns["ID"]!;
        DataColumn childCol = ds.Tables["Contact"]!.Columns["PersonID"]!;
        DataRelation rel = new DataRelation("PersonHasContacts", parentCol, childCol);
        ds.Relations.Add(rel);
        DataRow personRow = personTable.NewRow();
        personRow[1] = "Wolfgang";                          // column 1 is FirstName
        personRow["LastName"] = "Beer";
        personTable.Rows.Add(personRow);
        DataRow contactRow = contactTable.NewRow();
        contactRow["FirstName"] = "Wolfgang";
        contactRow["LastName"] = "Beer";
        contactRow["Phone"] = "08123456789";
        contactRow["PersonID"] = (long)personRow["ID"];     // defines the relation
        contactTable.Rows.Add(contactRow);
```

*`ID` in **Listing 15.8** is `ReadOnly`, `Unique` and `AutoIncrement` from 1; adding the relation also creates a foreign-key constraint. `NewRow` makes a row not yet in the table; `GetChildRows("PersonHasContacts")` is the in-memory join. Book: Section 15.6.3.*

The States of a Row

```
public DataRowState RowState { get; }
```

```
public enum DataRowState {
    Added, Deleted, Detached, Modified, Unchanged
}
```

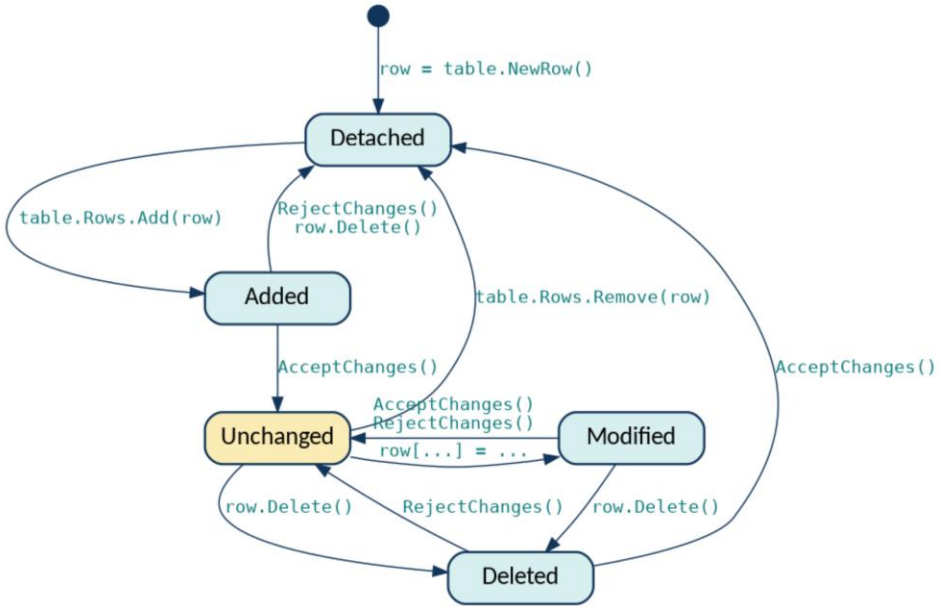


Figure 15.12 The state diagram of a DataRow. `NewRow` creates a *Detached* row; `Rows.Add` makes it *Added*; `AcceptChanges` makes *Added* and *Modified* rows *Unchanged* and removes *Deleted* ones; assigning a value makes an *Unchanged* row *Modified*; `Delete` makes it *Deleted*; `RejectChanges` returns *Modified* and *Deleted* rows to *Unchanged* and removes *Added* ones.

***Figure 15.12**: a DataSet remembers every change since it was filled. `AcceptChanges` makes the current values the original ones; `RejectChanges` restores them. A DataAdapter reads the states to decide between `INSERT`, `UPDATE` and `DELETE`. Book: Section 15.6.4.*

Worked Example 3: One Row Through Every State

Listing 15.9 The real output of `AdoBasics/RowStates.cs`: one `Service` row, its state and its three versions of the price after every step.

```
row = table.NewRow()                Detached   current=-      original=-      proposed=8000
table.Rows.Add(row)                 Added      current=8000   original=-      proposed=-
AcceptChanges()                     Unchanged  current=8000   original=8000   proposed=-
BeginEdit(); price = 8500           Unchanged  current=8000   original=8000   proposed=8500
EndEdit()                           Modified   current=8500   original=8000   proposed=-
RejectChanges()                     Unchanged  current=8000   original=8000   proposed=-
price = 9000; AcceptChanges()       Unchanged  current=9000   original=9000   proposed=-
row.Delete()                        Deleted    current=-      original=9000   proposed=-
RejectChanges()                     Unchanged  current=9000   original=9000   proposed=-
row.Delete(); AcceptChanges()       Detached   (gone: the table has 0 rows)
```

*In **Listing 15.9**: a new row has only a proposed value; between `BeginEdit` and `EndEdit` the state does not change; a deleted row keeps only its original, so the adapter can still say which row to delete. `RejectChanges` does not clear a `RowError`: call `ClearErrors`. Book: Sections 15.6.4–15.6.6.*

DataView: Sort, Find, Filter

Listing 15.10 The real output of `AdoBasics/DataViews.cs`: one view's sort, find, filter and row-state filter.

```
Sort = "LastName ASC"  (3 rows)
   1  Wolfgang  Beer        Unchanged
   3  Anna      Kournikova  Unchanged
   2  Maria     Sharapova   Unchanged
Find("Beer") = 0   Find("Kournikova") = 1   Find("Nobody") = -1
RowFilter = "FirstName <= 'K'"  (1 rows)
   3  Anna      Kournikova  Unchanged
RowStateFilter = Added | ModifiedCurrent  (2 rows)
   2  Mary      Sharapova   Modified
   4  Hendra    Wijaya      Added
RowStateFilter = ModifiedOriginal  (1 rows)
   2  Maria     Sharapova   Modified
ToTable(): a new table "Sorted" with 4 rows, first Wolfgang
```

*In **Listing 15.10**, `Find` does a binary search on the sort column and returns the position in the view, or −1. A view copies no rows. `RowFilter` is an expression language: gluing a visitor's text into it is as dangerous as gluing it into SQL. Book: Section 15.6.7.*

The Whole Laundry in Memory

Listing 15.11 From the output of `dotnet run -- laundry`: the seven tables with their constraints, and bill B0003 reached through its relations.

```
Outlet         3 rows  key id  constraints 1
Customer     256 rows  key id  constraints 1
Employee      14 rows  key id  constraints 1
Service       12 rows  key id  constraints 1
Bill        2953 rows  key id  constraints 5
Payment     3319 rows  key id  constraints 2
StatusLog   8802 rows  key id  constraints 3
relations: Customer_Bill, Employee_Bill, Service_Bill, Outlet_Bill, Bill_Payment, Bill_StatusLog,
Employee_StatusLog
Bill B0003: Ma'ruf Saputra, 2 x Bed cover, any size (per item) = Rp 70.000
  paid Rp 70.000 by cash at 2025-10-01 16:49:31
  2025-10-01 16:47:03  queued   by Dewi Lestari Putri
  2025-10-04 07:08:31  finished by Yulia Kartika Sari
  2025-10-05 07:11:16  departed by Dewi Lestari Putri
```

*In **Listing 15.11** every relation is also a foreign-key constraint; a DataTable has no `CHECK`, so `RowChanging` handlers enforce the payment methods. `DataTable.Compute` gave each outlet's September revenue, and SQLite agreed. Book: Section 15.6.8.*

The DataAdapter

- A **DataAdapter** connects a DataSet to a data source
  - **Fill**: fills the DataSet
  - **Update**: writes the changes back
- A DataAdapter uses Command objects
  - SelectCommand
  - InsertCommand
  - DeleteCommand
  - UpdateCommand

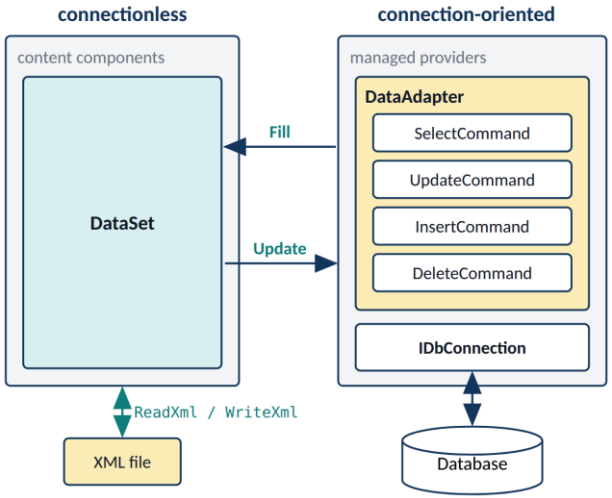


Figure 15.13 The DataAdapter between the connectionless and the connection-oriented worlds. Its four commands share one connection; `Fill` runs the `SelectCommand` and loads the DataSet; `Update` runs the insert, update and delete commands for the rows that changed.

***Figure 15.13**: `Fill` runs the `SelectCommand` and loads the DataSet; `Update` runs the insert, update and delete commands for the rows that changed. `Fill` opens a closed connection and closes it again. Book: Sections 15.7.1 and 15.7.2.*

The Round Trip, with a Conflict Check

Listing 15.12 From `AdapterRoundTrip.MySql.cs`: `Fill`, a CommandBuilder, the two row events, and a change in memory. Not run in this book (no MySQL, no NuGet).

```
var adapter = new MySqlDataAdapter("SELECT id, description, price FROM Service", con);
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;   // bring the primary key too
var builder = new MySqlCommandBuilder(adapter);                 // writes INSERT/UPDATE/DELETE for us
adapter.UpdateBatchSize = 50;                                   // many rows per round trip

adapter.RowUpdating += (_, e) => Console.WriteLine($"updating {e.StatementType} {e.Row["id"]}");
adapter.RowUpdated += (_, e) => Console.WriteLine($"updated  {e.StatementType} {e.Row["id"]}: {e.RecordsAffected}
row(s)");

var ds = new DataSet("Laundry");
adapter.Fill(ds, "Service");                    // opens, reads, closes: the connection is not kept
DataTable service = ds.Tables["Service"]!;
Console.WriteLine($"{service.Rows.Count} services loaded; connection state now {con.State}");

service.Rows.Find("S0003")!["price"] = 6500m;   // a change in memory only
Console.WriteLine(builder.GetUpdateCommand().CommandText);
```

*The generated `UPDATE` in **Listing 15.12** compares every original value; if nobody's row matches, `Update` throws `DBConcurrencyException`: optimistic concurrency. `Microsoft.Data.Sqlite` has no DataAdapter. DataSets also read and write XML, schemas and DiffGrams. Book: Sections 15.7.3–15.7.5 and 15.8.*

A Stored Procedure

> **DEFINITION.** A **stored procedure** is a named program written in the database's own procedural SQL, stored *in* the database and run *by* the server when a client calls it with `CALL name(arguments)`.

Listing 15.13 From `create_bill_with_payment.mysql.sql`: the procedure's `catch`, its transaction, and the next id read under a lock. Not run in this book.

```
    -- Any SQL error below: undo everything this call did, then report the error unchanged.
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        RESIGNAL;
    END;

    START TRANSACTION;
    -- FOR UPDATE locks what it reads, so two cashiers cannot compute the same next id.
    SELECT MAX(id) INTO v_max FROM Bill FOR UPDATE;
    SET p_bill_id = CONCAT('B', LPAD(COALESCE(CAST(SUBSTRING(v_max, 2) AS UNSIGNED), 0) + 1, 4, '0'));
```

*In **Listing 15.13**, `EXIT HANDLER … ROLLBACK; RESIGNAL` is the `catch`, `SIGNAL SQLSTATE '45000'` the `throw`, and `FOR UPDATE` stops the phantom. For: one round trip, one implementation, `GRANT EXECUTE` only. Against: a weaker language, a dialect per database, and SQLite has none. Book: Sections 15.10.1 and 15.10.2.*

Calling It from ADO.NET

Listing 15.14 From `CallProcedure.MySql.cs`: `CommandType.StoredProcedure`, one parameter per procedure parameter, and an output parameter for the new bill id. Not run in this book.

```
using var cmd = new MySqlCommand("create_bill_with_payment", con);
cmd.CommandType = CommandType.StoredProcedure;          // CommandText is a procedure name, not SQL
cmd.Parameters.AddWithValue("p_customer", "C0001");
cmd.Parameters.AddWithValue("p_service", "S0001");
cmd.Parameters.AddWithValue("p_unit", 5);
cmd.Parameters.AddWithValue("p_rack", "A01");
cmd.Parameters.AddWithValue("p_outlet", "O01");
cmd.Parameters.AddWithValue("p_employee", "E0004");
cmd.Parameters.AddWithValue("p_now", new DateTime(2026, 9, 30, 21, 0, 0));
cmd.Parameters.AddWithValue("p_method", "cash");
cmd.Parameters.AddWithValue("p_amount", 20000m);
cmd.Parameters.AddWithValue("p_reference", DBNull.Value);
var billId = new MySqlParameter("p_bill_id", MySqlDbType.String, 5) { Direction = ParameterDirection.Output };
cmd.Parameters.Add(billId);
```

*In **Listing 15.14**, `CommandText` is a procedure name, not SQL. A stored procedure and a repository method are two places for the same unit of work: what matters is that there is exactly one. Book: Section 15.10.3.*

Worked Example 4: SQL Injection

Step 1. The program glues the visitor's text between quotes:

```
"SELECT id, name FROM Customer WHERE name = '" + input + "'"
```

Step 2. The visitor types `x' OR '1'='1`, and the server parses:

```
SELECT id, name FROM Customer WHERE name = 'x' OR '1'='1'
```

Step 3. The condition is Equation (15.4), true for every row:

$$(name = \text{'x'}) \vee (\text{'1'} = \text{'1'}) \tag{15.4}$$

*The visitor's first quote closed the programmer's string, and the programmer's last quote closed the visitor's. Injection is not about SQL; it is about **gluing**. Book: Section 15.11.1.*

Worked Example 4: What It Did

Listing 15.15 The real output of `php/demo/injection.php` on a copy of the Laundry database: a glued query, the same query prepared, and a glued `UPDATE`.

```
The visitor types: Nabila Rahman
  glued     SELECT id, name FROM Customer WHERE name = 'Nabila Rahman'     -> 1 rows
  prepared  SELECT id, name FROM Customer WHERE name = ?                   -> 1 rows
The visitor types: x' OR '1'='1
  glued     SELECT id, name FROM Customer WHERE name = 'x' OR '1'='1'      -> 256 rows
  prepared  SELECT id, name FROM Customer WHERE name = ?                   -> 0 rows

glued UPDATE meant for one customer: customers with zero credit went from 232 to 256
```

The glued `SELECT` leaked all 256 customers; the prepared one found none, because nobody is named `x' OR '1'='1`.

The glued `UPDATE`, meant for one customer, set the deposits of all 24 who had one to zero.

*In **Listing 15.15**, a glued `UPDATE` lets the stranger choose which rows change. The same attack works on a `DataTable.Select` expression; doubling quotes helps, but not building the expression at all is the principle. Book: Sections 15.11.1 and 15.11.2.*

Glued Against Parameterized

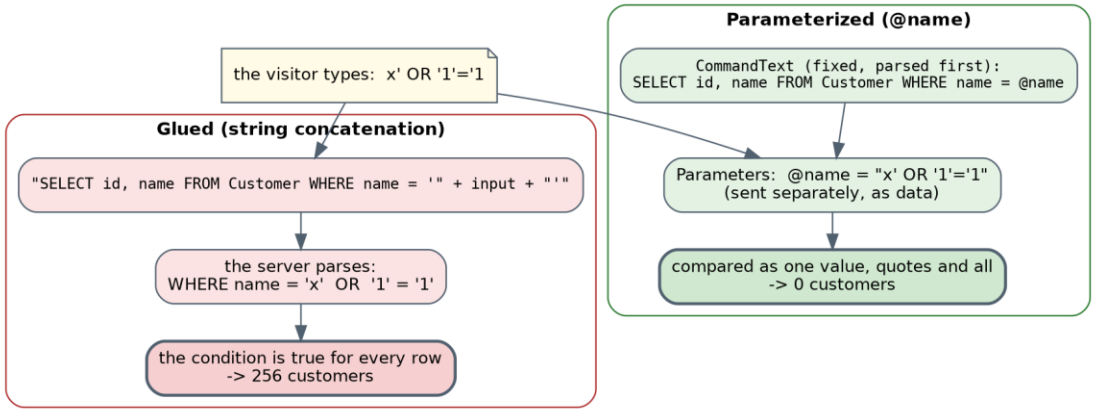


Figure 15.14 Glued against parameterized. When strings are glued, the visitor's text becomes part of the statement the server parses; when the statement is parameterized, the text is sent separately and can only ever be compared, as one value.

***Figure 15.14**: with parameters the text reaches the database first and its meaning is decided; the value arrives afterwards, as data, and can only be compared. There is nothing to escape and nothing to forget. Book: Section 15.11.3.*

What Parameters Cannot Do

Identifiers — table and column names, `ORDER BY` direction: use an allow-list, as `Count(table)` does.

`LIKE` **patterns** — `%` and `_` are wildcards even in a parameter: escape them and declare `ESCAPE '!'`.

Other languages — a `RowFilter`, XPath, a shell command: only values that passed a pattern such as `^O[0-9]{2}$`.

Defense in depth — least privilege, no SQL in error messages, validation at the boundary.

Parameters carry values; the three things that are not values each need their own defense. Book: Sections 15.11.4 and 15.11.5.

A Repository

> **DEFINITION.** A **repository** is an object that gives the rest of the application access to stored data through methods named in the language of the business, and hides how and where the data is stored. Code above the **data layer** never sees a connection, a command, a DataSet or a line of SQL.

Listing 15.16 The members of `IBillRepository`: everything the pages may ask of storage.

```
public interface IBillRepository
{
    BillView? FindBill(string id);
    IReadOnlyList<BillView> RecentBills(string? outletId, int limit);
    IReadOnlyList<CustomerRow> SearchCustomers(string text, int limit = 20);
    string CreateBillWithFirstPayment(NewBill bill);
    void AddPayment(string billId, string method, decimal amount, string? reference, string now);
    string Advance(string billId, string employeeId, string now);
    int Count(string table);
}
```

Listing 15.16 *buys pages that cannot misbehave, one test program for every implementation, and freedom to change storage in one line of* `Program.cs`*.* `BillView.Balance` *is Equation (15.3). Book: Section 15.12.1.*

The Data Layer of Profitina Laundry

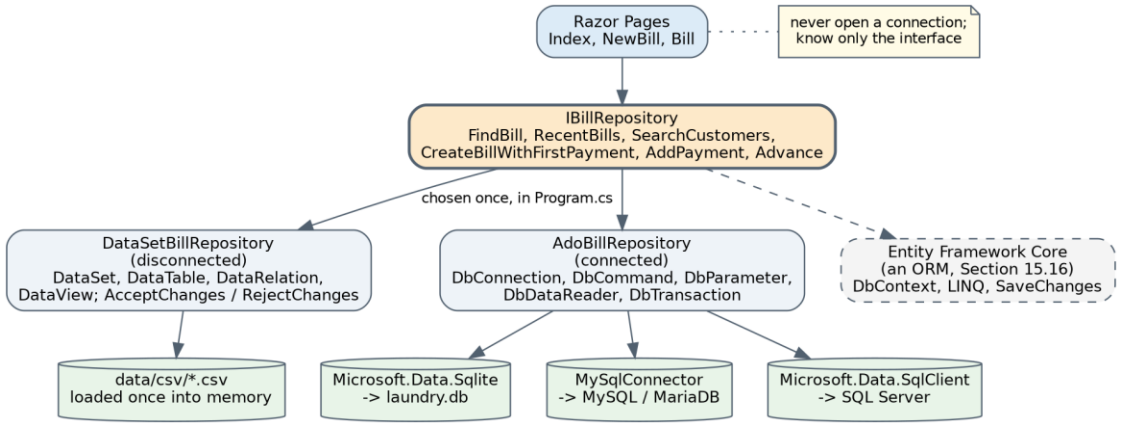


Figure 15.15 The Laundry data layer: Razor Pages know only `IBillRepository`, behind which `DataSetBillRepository` keeps the database in memory, `AdoBillRepository` reaches a real database through whichever provider it is given, and Entity Framework Core is a third possibility. Which one the pages receive is decided once, in `Program.cs`.

***Figure 15.15**: `DataSetBillRepository` keeps the database in memory with `AcceptChanges` as commit and `RejectChanges` as rollback; `AdoBillRepository` reaches any provider, named in one small file compiled only with `-p:UseSqlite=true`. Book: Sections 15.12.2 and 15.12.3.*

Worked Example 5: Rollbacks, Counted

Listing 15.17 The first ten of the fifteen checks of `Laundry.Tests` against `DataSetBillRepository`.

```
DataSet repository (disconnected, loaded from data/csv)
  PASS  find B2953: Lestari Mahendra, Dry cleaning - dress/kebaya (per item), charged Rp 45.000, paid Rp 45.000,
balance Rp 0
  PASS  counts before: Bill 2953, Payment 3319, StatusLog 8802
  PASS  create bill + first payment: B2954, 5 kg x Rp 8.000 = Rp 40.000, paid Rp 20.000, balance Rp 20.000
  PASS  three rows written: Bill 2954, Payment 3320, StatusLog 8803
  PASS  create bill, no payment: B2955, charged Rp 70.000, nothing paid
  PASS  rollback: payment larger than the charge: A first payment must be between Rp 0 and the charge, Rp 40.000.
  PASS  rollback: bad method after bill and log were written: Payment method 'gopay' is not one of: cash, qris,
transfer, credit.
  PASS  rollback: unknown customer: ForeignKeyConstraint Customer_Bill requires the child key values (C9999) to
exist in the parent table.
  PASS  counts after three refusals: Bill 2955, Payment 3320, StatusLog 8804
```

*In **Listing 15.17** the over-payment was refused before anything was written, the bad method after the bill and its log were added, the unknown customer by the foreign key. After three refusals the counts are exactly those after the two good bills. Book: Section 15.12.4.*

The Pages over the Repository

Profitina Bills New bill Contacts (DataView)

Latest bills

Outlet O01 Show

Bill	Arrived	Outlet	Customer	Service	Units	Charged	Paid	Balance	Status
B2953	2026-09-30 20:49:43	O01	Lestari Mahendra	Dry cleaning - dress/kebaya (per item)	1	Rp 45.000	Rp 45.000	Rp 0	queued
B2949	2026-09-30 15:36:01	O01	Zahra Halim	Wash, dry & fold (per kg)	7	Rp 56.000	Rp 28.000	Rp 28.000	queued
B2948	2026-09-30 13:56:56	O01	Joko Mahendra	Ironing only (per kg)	6	Rp 36.000	Rp 36.000	Rp 0	queued
B2947	2026-09-30 13:39:03	O01	Maya Santoso	Ironing only (per kg)	7	Rp 42.000	Rp 42.000	Rp 0	queued
B2946	2026-09-30 10:27:55	O01	Aisyah Wijaya	Wash, dry & fold (per kg)	5	Rp 40.000	Rp 0	Rp 40.000	queued
B2945	2026-09-30 09:53:44	O01	Nadia Umi Susanto	Bed cover, any size (per item)	1	Rp 35.000	Rp 35.000	Rp 0	queued
B2942	2026-09-29 20:07:05	O01	Rudi Lestari	Express wash & fold, 1 day (per kg)	4	Rp 48.000	Rp 0	Rp 48.000	finished
B2939	2026-09-29 08:25:30	O01	Farah Wijaya	Ironing only (per kg)	3	Rp 18.000	Rp 18.000	Rp 0	finished
B2936	2026-09-28 19:16:08	O01	Ayu Lina Santoso	Blanket (per item)	2	Rp 50.000	Rp 50.000	Rp 0	queued
B2935	2026-09-28 17:47:32	O01	Nanda Ramadhan	Bed cover, any size (per item)	1	Rp 35.000	Rp 35.000	Rp 0	queued
B2933	2026-09-28 11:22:04	O01	Oki Kusuma	Wash, dry & fold (per kg)	7	Rp 56.000	Rp 28.000	Rp 28.000	finished
B2928	2026-09-27 20:48:07	O01	Salsabila Susanto	Wash, dry & fold (per kg)	9	Rp 72.000	Rp 72.000	Rp 0	departed

Figure 15.16 The bills page filtered to outlet O01, after the session's first successful POST: B2954, Nabila Rahman, 5 kg of wash-dry-fold, Rp 40.000 charged, Rp 20.000 paid, Rp 20.000 still owed.

***Figure 15.16**: Razor Pages over `IBillRepository` on port 9501, after Post/Redirect/Get. A refused POST shows the repository's sentence with the cashier's input kept; the Web Forms `MyADO2.aspx` becomes a Razor Page with no postback and no view state. Book: Sections 15.12.5 and 15.12.6.*

Four Tracks Side by Side

Table 15.5 The Laundry data layer in four languages: six of the ten rows.

Idea	C# / ADO.NET	PHP / PDO	Java / JDBC	Python / `sqlite3`
Connection	`DbConnection`	`PDO`	`Connection`	`sqlite3.Connection`
Placeholder	`@name`	`?`, `:name`	`?`	`?`, `:name`
Read rows (connected)	`DbDataReader.Read()`	`foreach` over `PDOStatement`	`ResultSet.next()`	iterate the cursor
Begin	`BeginTransaction()`	`BEGIN IMMEDIATE` / `beginTransaction()`	`setAutoCommit(false)`	`BEGIN IMMEDIATE`
End	`Commit()` / `Rollback()`	`COMMIT` / `ROLLBACK`	`commit()` / `rollback()`	`COMMIT` / `ROLLBACK`
Tests here	15 checks, passed	15 checks, passed	15 checks, passed	10 tests, passed

***Table 15.5**: the SQL is the same in all four; the PHP, Java and Python tests ran against the same SQLite database, which is the evidence that the C# SQL is right. Java's `CachedRowSet` is its DataTable. Book: Sections 15.13–15.15 and 15.17.*

Entity Framework Core

Listing 15.18 From the unit of work with EF Core: LINQ for the price and the next id, child objects added through navigation properties, one `SaveChanges`. Not compiled or run in this book.

```
        using var tx = db.Database.BeginTransaction();          // so that MAX(id) is read inside it
        Service service = db.Services.Single(s => s.Id == b.ServiceId);
        string max = db.Bills.Max(x => x.Id);
        string id = "B" + (int.Parse(max[1..]) + 1).ToString("D4");
        bill.Log.Add(new StatusLogRow { Status = "queued", Changed_at = b.Now, Employee_id = b.EmployeeId });
        if (b.PayAmount > 0)
            bill.Payments.Add(new PaymentRow { Paid_at = b.Now, Method = b.PayMethod ?? "",
                                               Amount = b.PayAmount, Reference = b.PayReference });
        db.Bills.Add(bill);
        db.SaveChanges();                                       // three INSERTs
        tx.Commit();
        return id;
```

***Listing 15.18**: an ORM maps classes to tables; LINQ becomes parameterized SQL; `SaveChanges` sends the three `INSERT`s in one transaction. Dapper sits between: you write the SQL, it maps the rows. EF Core is ADO.NET underneath. Book: Section 15.16 and Table 15.9.*

Profitina in Practice

One data layer: pages and API handlers ask for “this user’s dashboard” or “record this purchase”; they never hold a connection.

Units of work: a renewal paid from a balance, a QRIS purchase credited, a post with its counters — each one transaction.

Parameters, always, allow-lists for identifiers, and one SQL for PostgreSQL in production and SQLite in tests.

UU 27/2022: a data export is a read across every table; an account purge is one transaction that removes all or nothing.

The runnable case study is Profitina Laundry; what carries across is the shape. Book: Section 15.18.

Common Mistakes

Gluing text into SQL or a `RowFilter`; a password in source; connecting as `sa` or `root`; forgetting to close.

A command without its transaction; the next id read outside the transaction; SQLite's foreign keys left off; a deferred `BEGIN` for read-then-write.

Casting a NULL; `HasErrors` without `ClearErrors`; trusting an inferred XML schema; looking for a DataAdapter in `Microsoft.Data.Sqlite`.

The database's error shown to a visitor; `LIKE` with the visitor's `%`; data code called from the page.

The book pairs each mistake with its fix. Book: Section 15.19.

Summary: Five Things to Remember

1. ADO.NET is one interface with a provider per database and two halves (**Figure 15.2**); Equation (15.1) says why web pages mostly use the connected one.

2. Open late, close always (**Pseudocode 15.1**); values travel only as parameters (**Listing 15.3**).

3. A transaction makes several writes one act (**Figure 15.7**); the next id needs a write lock or an auto-increment key.

4. The DataSet is an in-memory relational database that remembers every change in row states and versions (**Figure 15.12**).

5. SQL injection is gluing (Equation (15.4)); a repository hides the data layer behind the language of the shop (**Figure 15.15**).

Book: Section 15.20; Exercises 15.1–15.12 with solutions in Sections 15.21 and 15.22.

Next Time

Chapter 16 — Evaluation and Discussion IV: One Complete ASP.NET Application, and the Whole Book.

Chapters 13, 14 and 15 put together in one application, and problems that range over the whole book.

Before then: work Exercises 15.2, 15.5, 15.6 and 15.8, and run one track's tests yourself.

The grid, the validated front desk and the data layer meet in one application. Book: Chapter 16.

Lecture 16 — Evaluation and Discussion IV: One Complete ASP.NET Application, and the Whole Book

Course: Web Programming (EF234301)

Book: Irfan Subakti, *Web Programming: From HTML to ASP.NET Core, with PHP, Java and Python*, Chapter 16

Lecturer: Irfan Subakti — yifana@gmail.com

Today we build one complete website, the Laundry Counter, as six problems, work seven new problems on Chapters 13 to 15, and look back over the whole book. Book: Chapter 16, Sections 16.1–16.14.

What This Meeting Covers

1. A method and a map — three ASP.NET decisions, and every Web Forms idea in its Razor Pages form.

2. The project, Problems 16.1–16.6 — database, login, CRUD, design, evidence, hosting.

3. Data grids, Problems 16.7–16.8 — safe paging and sorting; the total that belongs in a footer.

4. Validation and state, Problems 16.9–16.10 — the server decides; one right home for each value.

5. The data layer, Problems 16.11–16.13 — $N+1$ questions, all or nothing, and the lost update.

6. The capstone and the whole book — the complete Profitina Laundry application, and where to go next.

Every problem follows the same order as in Chapters 4, 8 and 12: the problem, in our own words, design, the solution in three languages, line by line, a tempting wrong solution, and check yourself. Book: Section 16.1.

Where We Left Off

Chapters 1–4: HTTP, HTML, CSS, JavaScript and PHP. **5–8:** SQL, PDO, sessions, XML, JSP and JDBC, and the Laundry Desk. **9–12:** ASP.NET and Web Forms.

Chapter 13: data on the page. **Chapter 14:** validation and state. **Chapter 15:** ADO.NET, connected and disconnected. Today leans on these three hardest.

> **HOW TO USE THIS CHAPTER.** Before each problem, close the book for ten minutes and write down the tables you would touch, the addresses (URLs) of the pages, and one input that you expect to break a careless solution. Then read on.

Web Forms runs only on Windows; every problem here is solved in Razor Pages on .NET 10, and again in Java and in Python, so that three servers answer the same 37 checks. Book: Section 16.1.

Three ASP.NET Decisions

1. **The page model:** Razor Pages, the closest heir of Web Forms with the least ceremony; MVC and minimal APIs are the alternatives.
2. **Where the data layer ends:** one interface, `ILaundryStore`, with a disconnected `DataSetStore` and a connected `DbLaundryStore`; one line in `Program.cs` swaps them.
3. **Where each piece of state lives:** the grid in the query string, the user in an encrypted cookie, a message in TempData, recent bills in the session, facts in the data layer.

> **DEFINITION.** A *page model* is the class behind a Razor page: its *properties* hold what the page shows and receives, and its *handlers* (`OnGet`, `OnPost`) run for each kind of request. *Model binding* copies the request's fields into `[BindProperty]` properties; *model validation* then records every failure in `ModelState`.

These three decisions come on top of the eight-step method of Chapter 8. Book: Section 16.2.

From Web Forms to Razor Pages

Table 16.1 Web Forms ideas of Chapters 9 to 15, their Razor Pages form, and where they appear in the Laundry Counter.

Web Forms (Chapters 9 to 15)	Razor Pages on .NET 10 (this chapter)	Where in the Laundry Counter
`.aspx` page and `.aspx.cs` code-behind	`.cshtml` page and `.cshtml.cs` page model	every page
`Page_Load`, button click on a postback	handler methods `OnGet`, `OnPost`	`Login.cshtml.cs`
`GridView` with `AllowPaging`, `AllowSorting`	an HTML table in a loop, with sort and page in the query string	`Customers/Index`
`RequiredFieldValidator`	`[Required]`	`CustomerForm`, `BillForm`
`CustomValidator`	`IValidatableObject.Validate`	`BillForm`
`ValidationSummary`, `Page.IsValid`	`asp-validation-summary`, `ModelState.IsValid`	every form
view state (`__VIEWSTATE`)	none by default; hidden fields and the query string	the grid, the forms
`Application["key"]`	a singleton service	`ShopClock`, `TeachingSwitches`

Table 16.1 is the bridge from Chapters 9 to 15 to today; the book's full table has seventeen rows. Book: Section 16.2 and Table 16.1.

The Project

> **PROBLEM.** Using ASP.NET, build a CRUD website with login, registration and logout. It must work, and it should be beautiful. Write a report with the database design, the features, screenshots and a page-by-page explanation, and each member's share of the work. Deploy the website to a free host, and publish its source code on GitHub.

16.1 the database; **16.2** registration, login, logout; **16.3** create, read, update, delete; **16.4** beautiful; **16.5** the evidence and the report; **16.6** the repository and the hosting.

One website and one document, delivered in public, far easier to solve as a chain of six smaller problems than as one. Book: Section 16.4.

One Request Through the Laundry Counter

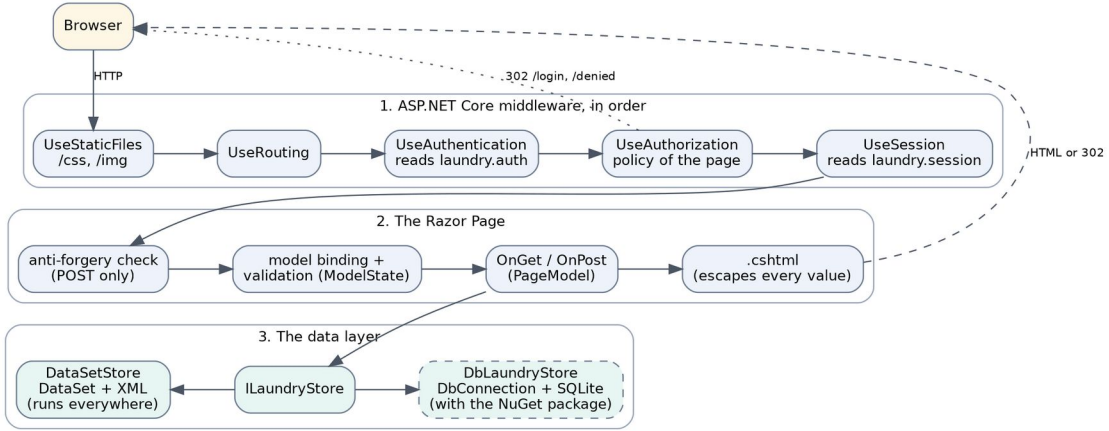


Figure 16.1 One request through the Laundry Counter: the middleware in the order `Program.cs` adds it, the Razor page, and the data layer. The dotted arrow is a refusal; the dashed box is the data layer that needs a NuGet package.

Figure 16.1: static files, routing, the login cookie turned into a user, the page's policy, the session, then for a `POST` the anti-forgery check, binding and validation, the handler, and the escaped markup; below, `ILaundryStore`. Every problem lives somewhere on this road. Book: Section 16.4.

Nine Features

Table 16.2 The nine features of the Laundry Counter, besides registration, login and logout. Every feature is an address that a test can call.

#	Feature	Who	Addresses	CRUD
F1	Customer grid: search, sort, page	staff	`GET /customers?q=&sort=&dir=&p=`	R
F2	Add a customer	staff	`GET`, `POST /customers/new`	C
F3	Edit a customer (the deposit: owner only)	staff	`GET`, `POST /customers/{id}/edit`	U
F4	Delete a customer (refused while bills exist)	staff	`POST /customers/{id}/delete`	D
F5	Price list; change a price	everyone; owner	`GET /`, `GET`, `POST /services/{id}/edit`	R, U
F6	Take in a bill, priced by the server	staff	`GET`, `POST /bills/new`	C
F7	Show a bill; move it to its next status	staff	`GET /bills/{id}`, `POST /bills/{id}/advance`	R, U, C
F8	A customer's own bills and balances	customer	`GET /my-bills`	R
F9	Money received per day and outlet	owner, supervisor	`GET /reports/revenue?month=&p=`	R

Table 16.2: every letter of CRUD appears at least twice, and every feature is an address that a test can call. Book: Section 16.4.

Two Business Rules on the Server

The amount charged is copied into the bill when it is written, Equation (16.1):

$$\text{charged} = \text{price}(\text{service}) \times \text{units} \tag{16.1}$$

What is still owed is computed, never stored, Equation (16.2):

$$\text{balance} = \text{charged} - \sum_{p \in \text{payments of the bill}} \text{amount}(p) \tag{16.2}$$

Equation (16.1) is why the bill form has no price field, and why a later change of price does not rewrite history; Equation (16.2) can never disagree with the payments. Book: Section 16.4.

Problem 16.1: Designing the Database

> **PROBLEM.** Give the database design of the website: the conceptual data model and the physical data model.

In our own words: two drawings and a proof. The proof is the part most write-ups forget: show that the rules are enforced by trying to break them.

Design: seven entities with the bill at the center; the physical model twice, as the SQL schema (a `CHECK` per rule) and as an ADO.NET `DataSet`.

> **DEFINITION.** A `DataRelation` joins a parent column to a child column and by default creates a unique constraint and a foreign key constraint, whose `DeleteRule` is `Cascade`, `SetNull`, `SetDefault` or `None`.

While `EnforceConstraints` is `true`, the `DataSet` refuses any row that breaks a key or a relation, as a database would. Book: Section 16.4.1.

The Physical Model as a DataSet

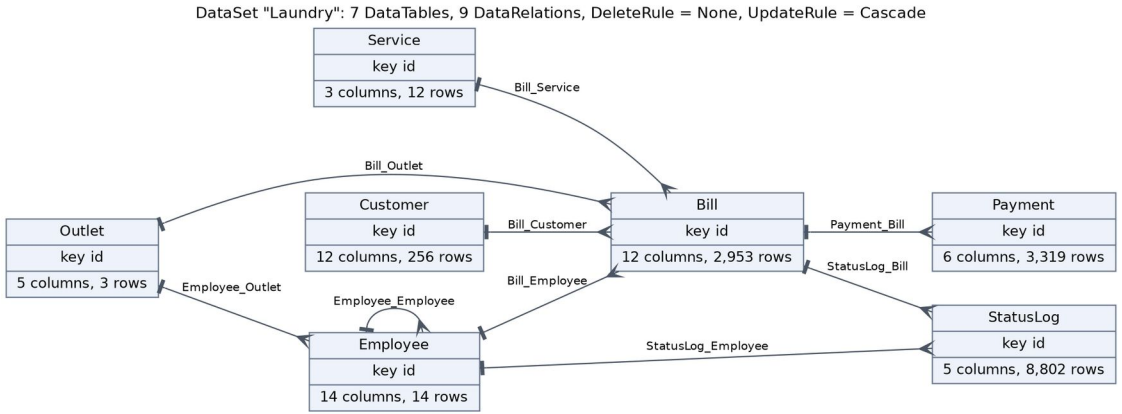


Figure 16.2 The physical model as an ADO.NET `DataSet`: seven tables with their keys and sizes, and nine relations named after the foreign keys of the SQL schema. The crow's foot marks the "many" end.

Figure 16.2: seven tables with typed columns, lengths and keys, and nine relations named exactly as the foreign keys of the SQL schema. Book: Section 16.4.1.

Problem 16.1: The Proof

Listing 16.1 `Relation` in `LaundryData.cs`: every relation's delete rule is chosen, and a changed key cascades into the children.

```
    private static void Relation(DataSet ds, string name, string parent, string child, string column,
Rule onDelete)
    {
        DataRelation r = ds.Relations.Add(name, ds.Tables[parent]!.Columns["id"]!,
ds.Tables[child]!.Columns[column]!);
        r.ChildKeyConstraint!.DeleteRule = onDelete;       // Rule.None: refuse to strand children
        r.ChildKeyConstraint.UpdateRule = Rule.Cascade;    // a changed key follows into the children
    }
```

```
1. a bill for customer C9999   REFUSED (InvalidConstraintException): ForeignKeyConstraint Bill_Customer
requires the child key values (C9999) to exist in the parent table.
2. a second customer C0001     REFUSED (ConstraintException): Column 'id' is constrained to be unique.
Value 'C0001' is already present.
3. a name of 51 letters        REFUSED (ArgumentException): Cannot set column 'name'. The value violates
the MaxLength limit of this column.
4. delete customer C0001       REFUSED (InvalidConstraintException): Cannot delete this row because
constraints are enforced on relation Bill_Customer, and deleting this row will strand child rows.
   bills 2953 -> 2953, payments 3319 -> 3319, log rows 8802 -> 8802
```

With Listing 16.1, four attempts are refused and nothing moves; a fifth, changing C0002's key, succeeds and her 21 bills follow it. Java and Python on SQLite agree. Book: Section 16.4.1.

Problem 16.1: The Tempting Default

Tempting: `ds.Relations.Add(...)` and stop. The default delete rule is `Cascade`; run with `--cascade`:

```
9 relations, delete rule Cascade: Employee_Outlet, Employee_Employee, Bill_Customer, Bill_Employee,
Bill_Service, Bill_Outlet, Payment_Bill, StatusLog_Bill, StatusLog_Employee
4. delete customer C0001       OK
   bills 2953 -> 2943, payments 3319 -> 3309, log rows 8802 -> 8772
```

Deleting C0001 silently deleted his ten bills, their payments and thirty rows of audit trail: fifty rows of history, one click.

> **CHECK YOURSELF.** A new table `BillPhoto(id, Bill_id, file)` holds photographs of stains. Which delete rule would you give `BillPhoto_Bill`, and why is the answer different from `Payment_Bill`?

Answer: `Cascade` suits a photograph, which means nothing without its bill; a payment is money that changed hands, so `Payment_Bill` stays `None`. The rule follows from what the child row is. Book: Section 16.4.1.

Problem 16.2: Registration, Login, Logout

> **PROBLEM.** The website must include registration, login and logout.

The unsaid requirement: every non-public page must *refuse* a visitor who is not logged in, or who is logged in with the wrong kind of account.

Design: staff use the database's accounts; customers register. Login gives two *claims*, the `kind` and the `role`, derived from the data. Passwords are PBKDF2-HMAC-SHA256 with 600,000 iterations.

The login cookie is encrypted, `HttpOnly` and `SameSite=Lax`, and lives for one shift; `AuthorizeFolder` puts whole folders under policies such as `Staff`. Book: Section 16.4.2.

A Login and the Next Request

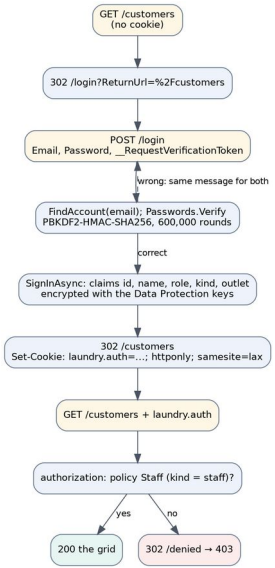


Figure 16.3 The login of a cashier and her next request. The dashed arrow is a refused login, which gives the same answer whether the e-mail or the password was wrong.

Figure 16.3: after the login the browser carries the cookie, and authentication turns it back into a user on every request. Book: Section 16.4.2.

The Login Handler

Listing 16.2 `OnPostAsync` in `Login.cshtml.cs`: the same work and the same message for a wrong e-mail and a wrong password, then the cookie, and a redirect only to a local address.

```
    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid) return Page();
        Account? a = store.FindAccount(Email.Trim());
        bool ok = Passwords.Verify(Password, a?.PasswordHash ?? Passwords.Dummy);  // same work either
way
        if (a == null || !ok)
        {
            ModelState.AddModelError("", "Wrong e-mail or password.");        // never say which one
            return Page();
        }
        HttpContext.Session.Clear();                          // nothing from before the login survives it
        await HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme,
            new ClaimsPrincipal(new ClaimsIdentity(claims,
CookieAuthenticationDefaults.AuthenticationScheme)));
        TempData["Flash"] = $"Welcome, {a.Name}.";
        string home = a.Kind == "staff" ? "/customers" : "/my-bills";
        return LocalRedirect(Url.IsLocalUrl(ReturnUrl) ? ReturnUrl! : home);   // never to another site
```

In Listing 16.2, an unknown e-mail still runs a full PBKDF2 check against `Passwords.Dummy`, so timing reveals nothing. `Session.Clear()` keeps nothing from before the login; `Url.IsLocalUrl` stops an open redirect. Logout is a `POST`. Book: Section 16.4.2.

Problem 16.2: Hidden Links Protect Nothing

Tempting: hide the “Customers” link and add no policy. A visitor who types the address:

```
anonymous GET /customers -> 200
  the page says: 256 customers, page 1 of 26.
  the menu shows a visitor a Customers link: no
```

With the policies in place, the same request:

```
anonymous GET /customers -> 302 /login
  the page says: (no customer data)
  the menu shows a visitor a Customers link: no
```

> **CHECK YOURSELF.** A supervisor at O02 may open the revenue report, but only for her own outlet. Does that rule belong in the policy, the page model, or the markup?

Answer: in the page model or the data layer, because it depends on the rows asked for; filtering in the markup would still send the other outlets’ figures to the browser. Book: Section 16.4.2.

Problem 16.3: Create, Read, Update, Delete

PROBLEM. The website must provide all four CRUD operations: create, read, update and delete.

Listing 16.3 Part of `ILaundryStore.cs`: what the pages may ask about accounts and customers, and nothing more.

```
public interface ILaundryStore
{
    Account? FindAccount(string email);
    string RegisterCustomer(CustomerInput c, string passwordHash, string now);

    GridPage<CustomerRow> Customers(string? search, string sort, bool descending, int pageNo, int
pageSize);
    CustomerRow? Customer(string id);
    string AddCustomer(CustomerInput c, string now);
    bool UpdateCustomer(string id, CustomerInput c, decimal? newCredit);   // null: credit unchanged
    bool DeleteCustomer(string id);                                        // false: he still has bills
```

Three layers: pages, the interface Listing 16.3, the store. Every change is Post/Redirect/Get. `DeleteCustomer` *returns* `false` *when* `Rule.None` *refuses;* `NewBill` *prices by Equation (16.1); the* `DataSet` *saves under one lock and rolls back with* `RejectChanges`*. Book: Section 16.4.3.*

Problem 16.3: Overposting

Tempting: bind the whole entity. A cashier posts the edit form of C0005 plus one extra field, `Credit=5000000`:

```
before: C0005 Lukman Santoso 0812-0000-2062 Rp 0
the cashier posts the edit form plus Credit=5000000 -> 302 /customers
after:  C0005 Lukman Santoso 0812-0000-2062 Rp 5.000.000
```

On the correct site, which binds an *input model* with only the editable fields:

```
before: C0005 Lukman Santoso 0812-0000-2062 Rp 0
the cashier posts the edit form plus Credit=5000000 -> 302 /customers
after:  C0005 Lukman Santoso 0812-0000-2062 Rp 0
```

> **CHECK YOURSELF.** A visitor adds `Input.Credit=1000000` to his registration request. What happens, and would it change if `CustomerForm` had a hidden `Credit` property?

Answer: nothing, because `CustomerForm` has no `Credit`; with such a property model binding would fill it. Hiding the input protects nothing; leaving the property out does. Book: Section 16.4.3.

Problem 16.4: Beauty by Rules

One frame, the main action most visible, numbers aligned right, a label on every field, a phone layout, a visible focus, and text dark enough: the WCAG contrast rule, Equation (16.3), Equation (16.4) and Equation (16.5).

$$c_{\text{lin}} = \begin{cases} c/12.92 & \text{if } c \le 0.04045 \\ \left(\frac{c+0.055}{1.055}\right)^{2.4} & \text{otherwise} \end{cases} \tag{16.3}$$

$$L = 0.2126\,R_{\text{lin}} + 0.7152\,G_{\text{lin}} + 0.0722\,B_{\text{lin}} \tag{16.4}$$

$$\text{ratio} = \frac{L_1 + 0.05}{L_2 + 0.05} \tag{16.5}$$

The ratio runs from 1 to 21; level AA asks for at least 4.5 for normal text. The colors come from the official Profitina logo, darkened until they pass. Book: Section 16.4.4.

Worked Example 1: Body Text on White

`#1f2937`: channels 0.1216, 0.1608, 0.2157 become 0.0137, 0.0222, 0.0382 by Equation (16.3); $L_2 = 0.0216$ by Equation (16.4); ratio $1.05 / 0.0716 = 14.7$.

Listing 16.4 Lines from the output of `P164Contrast.cs`: the contrast of the text colors of `site.css`, and the tempting light grey.

```
body text                   #1f2937 on #ffffff  14.68:1  AA pass
secondary text              #4b5563 on #ffffff   7.56:1  AA pass
secondary text on the page  #4b5563 on #f3f6fb   6.98:1  AA pass
link / button               #1d4ed8 on #ffffff   6.70:1  AA pass
status: departed            #166534 on #dcfce7   6.49:1  AA pass
tempting: light grey        #aaaaaa on #ffffff   2.32:1  FAIL
```

Listing 16.4: the tempting light grey `#aaaaaa` is 2.32 to 1, half of what WCAG asks; on a cheap phone in the sun that text is not there. Java and Python printed the same eleven lines. Book: Section 16.4.4.

The Grid, Sorted by Credit

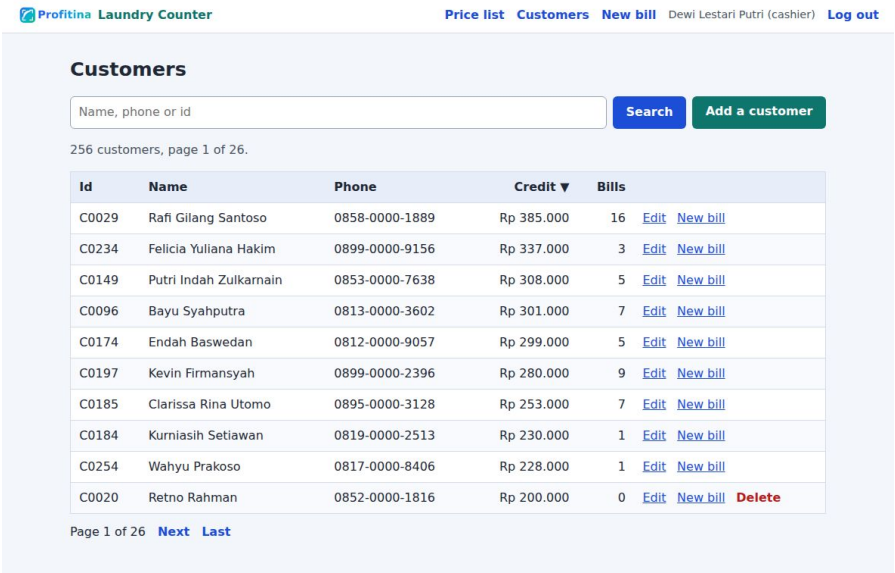


Figure 16.4 The customer grid of the ASP.NET Laundry Counter, sorted by credit, highest first. Only customers without bills have a Delete button.

Figure 16.4: numbers right-aligned in digits of equal width; `th[aria-sort]` draws the arrow from the same attribute a screen reader reads. Book: Section 16.4.4.

The Same Grid on a Phone

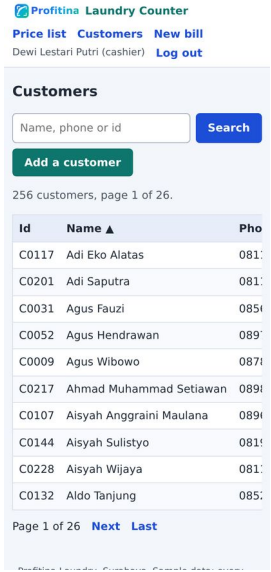


Figure 16.5 The same grid on a phone-sized screen: the menu wraps, and the grid scrolls inside its own box.

Figure 16.5: table cells do not wrap and the grid scrolls inside its own box; the menu wraps. Book: Section 16.4.4.

Problem 16.5: Evidence That Can Be Re-Run

> **PROBLEM.** Write a report with the database design, a list of features, screenshots, a page-by-page explanation, and each member's job, share and contributions.

```
PASS  A2   a visitor asking for /customers is sent to /login
PASS  A3   a wrong password and an unknown e-mail get the same answer
PASS  B2   every cookie set at login is HttpOnly and SameSite
PASS  B11  a cashier's edit is saved, but a posted Credit is ignored
PASS  C1   quantity 0 and rack Z99 are refused, each with its own message
PASS  C3   a valid bill is priced by the server: 5 kg x Rp 8.000
PASS  C6   a form without its anti-forgery token is refused
PASS  D4   the bill written earlier keeps the price it was charged
37 of 37 checks passed
```

An acceptance test over real HTTP ran against all three servers and the published build: `37 of 37 checks passed`, a sentence anyone can check. Screenshots alone cannot show a refusal. "Everyone did everything" cannot be checked; percentages that match the commits can. Book: Section 16.4.5 and Table 16.4.

The Login Page

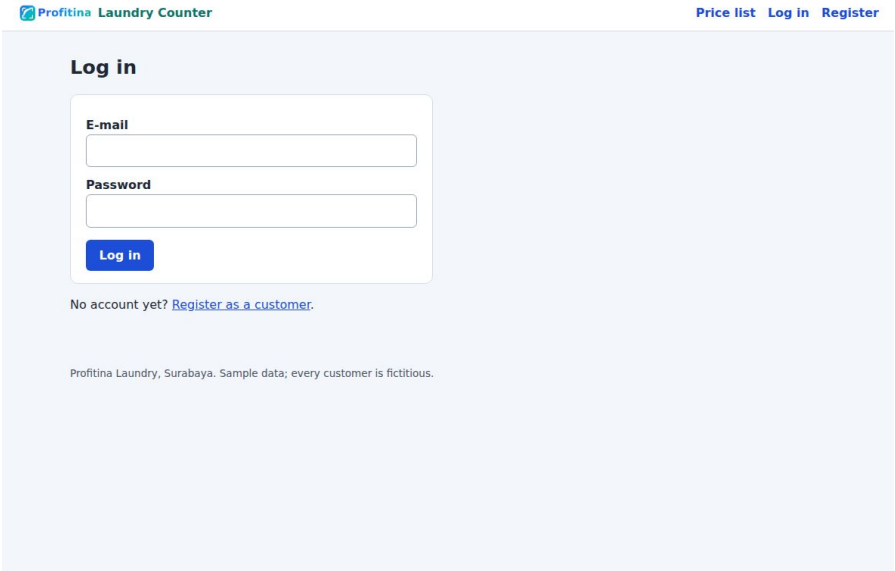


Figure 16.6 The login page. The same message answers a wrong e-mail and a wrong password.

Figure 16.6: taken by a script (Playwright), so it can be retaken after every change; the script also reports fields without a label: none. Book: Section 16.4.5.

One Bill in Full

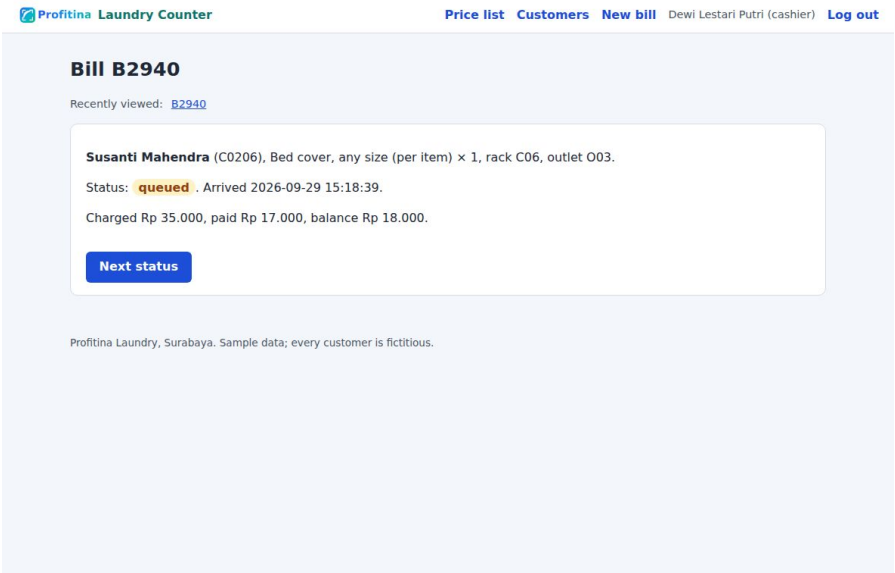


Figure 16.7 One bill (feature F7): customer, service, rack, status with its times, and the amounts of Equation (16.1) and Equation (16.2).

Figure 16.7: feature F7, with the amounts of Equation (16.1) and Equation (16.2) computed on the server. Book: Section 16.4.5.

Problem 16.6: The Repository and the Hosting

Design: no build output, live data or secrets in the repository; `dotnet publish -c Release`; settings from the environment (`Laundry__DataDir`); data and the *key ring* in folders that survive restarts.

```
login -> 302 /customers
restart, same key folder:
GET /customers with yesterday's cookie -> 200 (the grid)
restart, keys lost (a new, empty key folder):
GET /customers with yesterday's cookie -> 302 /login
```

Tempting: deploy the laptop as it is, and users are "thrown out" at every restart; or commit the keys and `.env`, and a secret is public forever.

The key ring encrypts the login cookie, the anti-forgery tokens and TempData; lose it and every cookie is unreadable. Java keeps sessions in memory; Flask needs a fixed `LAUNDRY_SECRET`. Book: Section 16.4.6.

Problem 16.7: Paging and Sorting Safely

PROBLEM. Ten customers per page, searchable, sortable by id, name, phone or credit; page, search and sort arrive in the address. Every combination of values must give a sensible page, never an error or a leak.

$$P = \max\left(1, \left\lceil \frac{n}{s} \right\rceil\right) = \max\left(1, \left\lfloor \frac{n+s-1}{s} \right\rfloor\right) \tag{16.6}$$

$$p' = \min\left(\max\left(p, 1\right), P\right) \tag{16.7}$$

$$\text{offset} = \left(p' - 1\right) s \tag{16.8}$$

Equation (16.6) with $s = 10$: $n = 10$ gives 1 page, $n = 11$ gives 2, $n = 256$ gives 26. Equation (16.7) turns page 0 into 1 and page 99 into the last; Equation (16.8) makes page 26 skip 250 rows. Book: Section 16.5.1.

Values Compared, Names from a List

Listing 16.5 From `DataSetStore.Customers`: the sort column is taken from `SortColumns`, and the search text is compared in C#, never parsed.

```
            var view = new DataView(T("Customer"));
            string col = SortColumns.GetValueOrDefault(sort, "name");      // only names we wrote
ourselves
            view.Sort = $"{col} {(descending ? "DESC" : "ASC")}, id ASC";
            IEnumerable<DataRow> rows = view.Cast<DataRowView>().Select(v => v.Row);
            string s = (search ?? "").Trim();
            if (s.Length > 0)                                              // compared in C#, never parsed
                rows = rows.Where(r => S(r, "name").Contains(s, StringComparison.OrdinalIgnoreCase)
                                    || S(r, "phone").Contains(s)
                                    || S(r, "id").Equals(s, StringComparison.OrdinalIgnoreCase));
```

In Listing 16.5 an unknown sort becomes `name`, and `id ASC` breaks ties. The Java and Python versions use placeholders for the search and for `LIMIT` and `OFFSET`. The query value is `p`, not `page`, which Razor Pages reserves. Book: Section 16.5.1.

Problem 16.7: The Tempting Filter

```
  sort 'name; DROP', page 99: 256 customers, page 26 of 26
      C0221  Yusuf Permana              0881-0000-7947  Rp 0
      C0076  Yusuf Rafi Sugiarto        0896-0000-3034  Rp 0
      C0091  Zahra Halim                0815-0000-5243  Rp 0
The tempting way: the visitor's text inside a filter or a sort expression:
  RowFilter with q = Wijaya                -> 8 rows
  RowFilter with q = Ma'ruf                -> SyntaxErrorException: Syntax error: Missing operand after
'ruf' operator.
  RowFilter with q = zzz' OR name <> '     -> 256 rows
  Sort = name; DROP                        -> IndexOutOfRangeException: Cannot find column name; DROP.
```

Glued into a `RowFilter`, `Ma'ruf` crashes the page and `zzz' OR name <> '` returns everyone; in SQL, an `ORDER BY` built from text leaks a hash one yes-or-no at a time: *blind* injection.

> **CHECK YOURSELF.** A search for `C0258` finds a customer whose name has no "C0258". Which part of the condition found him, and why is it an equality, not a `LIKE`?

The correct grid: `sort 'name; DROP'` becomes `name`, page 99 becomes 26. Answer: the id part, `upper(id) = upper(?)`; an id is a key, and `LIKE` would return noise. Book: Section 16.5.1.

Problem 16.8: A Grid with a Footer

> **PROBLEM.** Money received in a month, one row per day, one column per outlet and a total, ten days per page. Below the rows the grid shows totals. Which totals, computed from what?

With $r_{d,o}$ the money received on day d at outlet o, D_p the days on page p and D all the days, Equation (16.9):

$$\text{PageTotal}_o = \sum_{d \in D_p} r_{d,o} \qquad \text{MonthTotal}_o = \sum_{d \in D} r_{d,o} \tag{16.9}$$

The two are equal only when $D_p = D$, so the report shows both, each labeled with what it sums; the month total never comes from the rows on the screen. Book: Section 16.5.2.

September 2026 in Three Pages

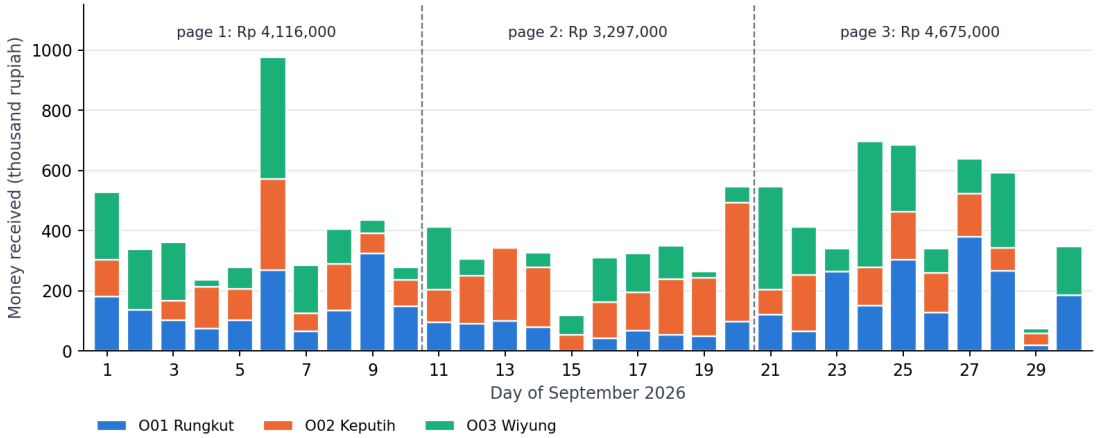


Figure 16.8 Money received in September 2026 per day and outlet, with the three pages of the report. The three page totals add up to the month total, Rp 12,088,000.

Figure 16.8: the three page totals are visibly different, and together they make the month. Book: Section 16.5.2.

Worked Example 2: Two Footer Rows

```
Day                   O01           O02           O03         Total
2026-09-21      Rp 120.000     Rp 84.000    Rp 343.000    Rp 547.000
2026-09-28      Rp 265.000     Rp 77.000    Rp 251.000    Rp 593.000
2026-09-29       Rp 18.000     Rp 40.000     Rp 17.000     Rp 75.000
2026-09-30      Rp 186.000           Rp 0    Rp 160.000    Rp 346.000
Page total    Rp 1.880.000    Rp 949.000  Rp 1.846.000  Rp 4.675.000
Month total   Rp 4.088.000  Rp 3.837.000  Rp 4.163.000 Rp 12.088.000
Compute(SUM(amount)) over September: Rp 12.088.000
```

By hand, O02 on page 3: $84+186+0+128+158+132+144+77+40+0=949$ thousand rupiah.

Tempting: sum the rows as they are drawn (`RowDataBound`): the "month total" becomes Rp 4.675.000, less than two fifths of the truth.

The month total, Rp 12.088.000, equals an independent `SUM(amount)`. The average per day is taken over all 30 days: $12,088,000/30=402,933$ *rupiah. Book: Section 16.5.2.*

The Revenue Report

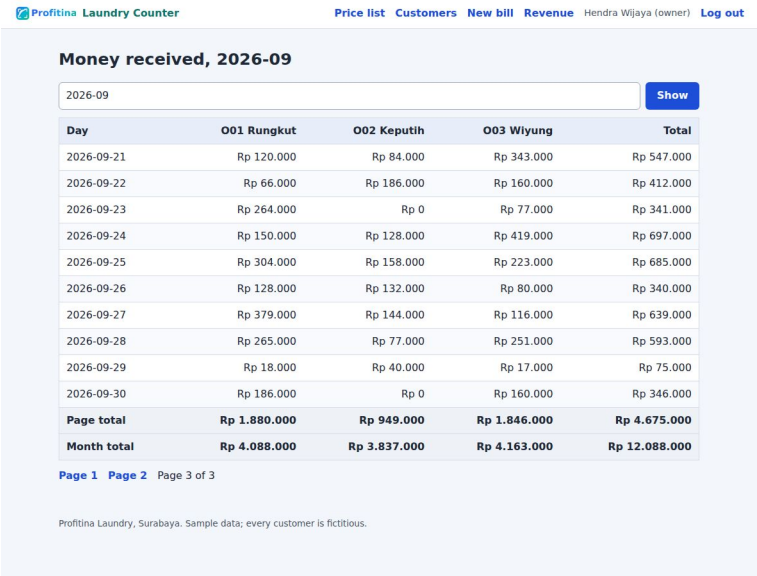


Figure 16.9 The revenue report, page 3 of September 2026 (feature F9). Two footer rows, each saying what it sums.

Figure 16.9: page 3 of September 2026; a month that does not exist is answered with 400, Bad Request. Book: Section 16.5.2.

Problem 16.9: Validation the Browser Cannot Switch Off

Listing 16.6 From `BillForm.cs`: two attributes that check one field each, and `Validate`, which needs the data.

```
    [Range(1, 100, ErrorMessage = "The quantity must be a whole number from 1 to 100."), Display(Name =
"Quantity")]
    public int Unit { get; set; } = 1;

    [Required, RegularExpression(@"^[A-Ca-c](0[1-9]|1[0-9]|20)$", ErrorMessage = "A rack is A, B or C and
a number 01 to 20, like A07.")]
    public string Rack { get; set; } = "";
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        ILaundryStore store = context.GetRequiredService<ILaundryStore>();
        if (store.Customer(CustomerId) == null)
            yield return new ValidationResult($"There is no customer {CustomerId}.",
[nameof(CustomerId)]);
        if (!store.Services().Any(s => s.Id == ServiceId))
            yield return new ValidationResult($"There is no service {ServiceId}.", [nameof(ServiceId)]);
    }
```

The browser's `min`, `max`, `required` and `pattern` are a courtesy; Listing 16.6 is the law, and `if (!ModelState.IsValid) return Page();` is one line every request passes through. `Validate` runs only when every attribute passed. Book: Section 16.6.1.

The Server Refuses Both Fields

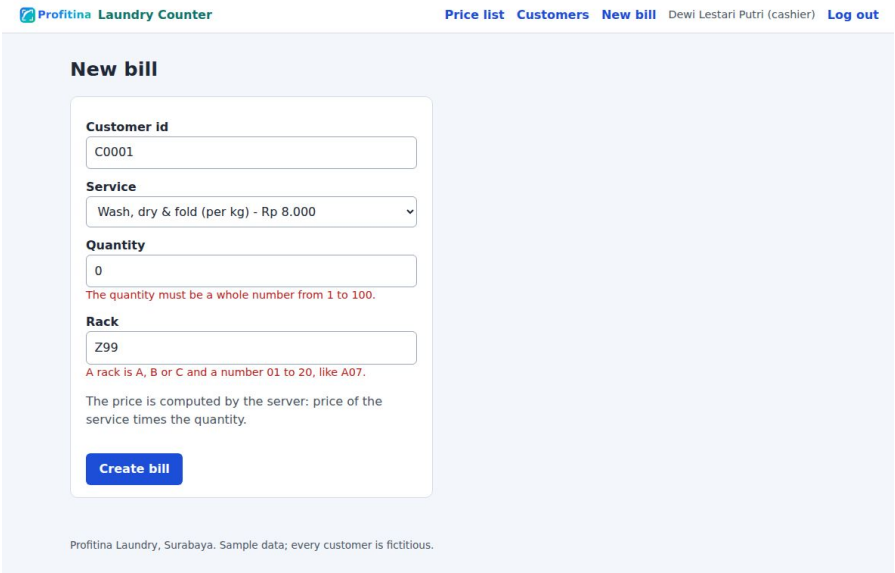


Figure 16.10 The new-bill form after the browser's own checks were switched off: the server refused both fields, each with its own message.

Figure 16.10: the browser's checks were switched off with `form.noValidate = true`, and a quantity of 0 and a rack of `Z99` were sent. Book: Section 16.6.1.

Problem 16.9: Minus Five Kilograms

Tempting: trust the browser, and a script sends what any script can send:

```
POST /bills/new with Unit=-5 and Rack=ZZ -> 302 /bills/B2954
  Charged Rp -40.000, paid Rp 0, balance Rp -40.000.
  rack ZZ
```

With the server's checks in place:

```
POST /bills/new with Unit=-5 and Rack=ZZ -> 200
  refused: The quantity must be a whole number from 1 to 100.; A rack is A, B or C and a number 01 to
20, like A07.
```

CHECK YOURSELF. Express service `S0002` is sold only up to 10 kg. Where does the rule go, and why not as an attribute?

Answer: in `Validate`, because the rule relates two fields; an attribute sees one property at a time. Equation (16.1) did its job faithfully, on nonsense. Book: Section 16.6.1.

Problem 16.10: Where Does Each Value Live?

PROBLEM. Show, at the top of every staff page, the last five bills *this* user has opened. Then list every piece of state the Laundry Counter keeps, and justify where each one lives.

DEFINITION. *Session state* is a small dictionary kept by the server for each visitor, who holds only its key in a cookie. *Application state* is shared by all visitors for the life of the process: a singleton service, or a `static` field.

One visitor, one working session, harmless if lost, private: the recent bills are session state, kept as one string such as `B2954,B0001`. Book: Section 16.6.2.

The State of the Laundry Counter

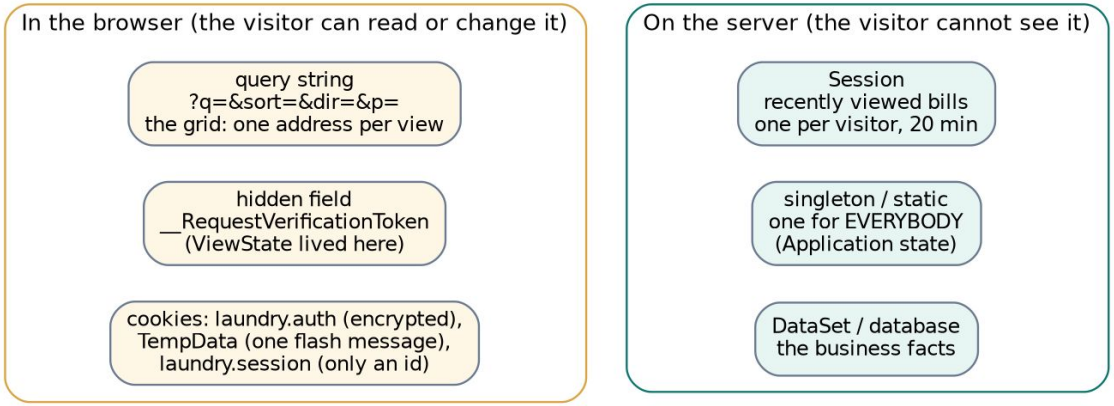


Figure 16.11 Where the state of the Laundry Counter lives. What is in the browser can be read or changed by the visitor; what is on the server cannot.

Figure 16.11: what is in the browser can be read or changed by the visitor; what is on the server cannot. Book: Section 16.6.2.

One Right Home per Value

Table 16.3 Every piece of state of the Laundry Counter, where it lives, who owns it, how long it lives, and why that is the right place.

Value	Lives in	Owner and lifetime	Why there
grid page, sort, search	query string	this view; as long as the address	bookmarkable, shareable, harmless if edited
anti-forgery token	hidden field and cookie	this form; one form	must travel with the form it protects
who is logged in	encrypted cookie `laundry.auth`	this browser; one shift	read on every request; cannot be forged without the keys
“Customer C0258 added.”	TempData cookie	this browser; the next page only	a message for one redirect
recently viewed bills	session (server memory)	this visitor; 20 minutes idle	per visitor, not worth storing, must not be shared
the fixed clock, the switches	singleton service	everybody; the process	the same for everyone, read-only
customers, bills, payments	the data layer	the business; forever	the facts; must survive everything

Table 16.3: each value has an owner, a lifetime and a level of trust. Book: Section 16.6.2 and Table 16.5.

Problem 16.10: A Static List

Tempting: a `static List<string>` survives between requests, so it seems made for the job. It is application state, one list for everybody:

```
the cashier opened B0001 and B2950
the owner, who opened nothing, sees: Recently viewed: B2950 B0001
```

With the session, the same two browsers:

```
the cashier opened B0001 and B2950
the owner, who opened nothing, sees: (no recently viewed bills)
```

> **CHECK YOURSELF.** "Bills created today, all outlets", updated instantly without a query: session, singleton or database? What does a restart cost?

Answer: a singleton with `Interlocked.Increment`, rebuilt from the database at start-up; the database stays the truth. Book: Section 16.6.2.

Problem 16.11: One Question or Three Thousand

> **PROBLEM.** Compute how many bills are still in the shop and how much all customers still owe, by Equation (16.2) over all bills, connected and disconnected, and explain which way of asking is fast.

```
reader: 57 bills still in the shop
one Select per bill:   owed Rp 1.529.000, 2954 questions, 56 ms
GetChildRows per bill: owed Rp 1.529.000, 1 load, 10 ms
Compute:               charged Rp 144.213.000 - received Rp 142.684.000 = Rp 1.529.000
```

$$T_{N+1} \approx (N+1)\,t_q + N\,t_r \text{ while } T_1 \approx t_q + N\,t_r \tag{16.10}$$

57 bills in the shop; Rp 1.529.000 owed. By Equation (16.10) the difference is $N\,t_q$*: with a database server at 0.5 ms a question,* $2{,}953 \times 0.5$ *ms* ≈ 1.5 *s for one page. Book: Section 16.7.1.*

Problem 16.11: The $N+1$ Trap

Tempting: "for every bill, fetch its payments": correct, natural, and the slow side of Equation (16.10). The C# filter `$"Bill_id = '{b["id"]}'"` is also glued text.

Cure: `GetChildRows("Payment_Bill")` in the disconnected model; one `GROUP BY` or join in the connected model.

CHECK YOURSELF. "My bills" computes each bill's paid amount with a correlated subquery. Does it have the $N+1$ problem?

Answer: no; the subquery runs inside the database, in one query, with no round trip per row. $N+1$ is about round trips. Book: Section 16.7.1.

Problem 16.12: A Bill and Its Log Row

The log row names employee `E9999`, so the foreign key refuses the second write. With a unit of work:

```
before: 2953 bills, 8802 log rows; bills without a log row: 0
error: ForeignKeyConstraint StatusLog_Employee requires the child key values (E9999) to exist in the
parent table.
after:  2953 bills, 8802 log rows; bills without a log row: 0
```

Tempting, each write committed on its own (`--no-tx`):

```
before: 2953 bills, 8802 log rows; bills without a log row: 0
error: ForeignKeyConstraint StatusLog_Employee requires the child key values (E9999) to exist in the
parent table.
after:  2954 bills, 8802 log rows; bills without a log row: 1
```

CHECK YOURSELF. Recording a payment writes one `Payment` row. Does it need a transaction?

The rule: when several writes express one fact, put them in one transaction. A single statement is already atomic; a second statement makes a transaction necessary. Book: Section 16.7.2.

Problem 16.13: The Lost Update

PROBLEM. Dewi and Rizky open customer C0005 at the same moment. Dewi corrects the phone and saves; a minute later Rizky corrects the address and saves. What happens to Dewi's correction, and how must the save be written?

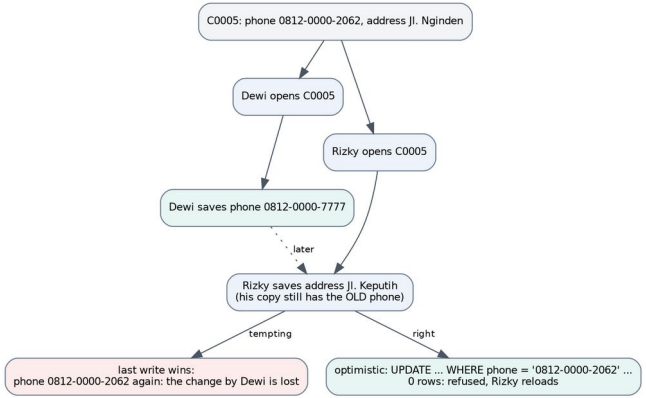


Figure 16.12 Two cashiers edit the same customer. Writing back the whole copy loses Dewi's change; checking the original values refuses Rizky's save instead.

Figure 16.12: each page is a disconnected copy of the row. Book: Section 16.7.3.

Worked Example 3: Optimistic Concurrency

With o_c the value of column c when the page opened and v_c its value now, the save succeeds exactly when Equation (16.11) holds:

$$\text{save succeeds} \Leftrightarrow \forall\, c : v_c = o_c \tag{16.11}$$

```
in the shop: Lukman Santoso, 0812-0000-2062, Jl. Nginden Intan Blok B No. 76, Rungkut, Surabaya
Dewi   saved
Rizky  REFUSED: somebody changed C0005 after you opened it; reload and try again
in the shop: Lukman Santoso, 0812-0000-7777, Jl. Nginden Intan Blok B No. 76, Rungkut, Surabaya
```

Tempting, "last write wins": both saves succeed, and the phone number is the old one again, although nobody typed it back.

In SQL: `UPDATE … WHERE id = @id AND phone = @old_phone …`; zero rows changed means "reload". A `version` column detects any change with a shorter `WHERE`, at the price of more refusals. Book: Section 16.7.3.

The Capstone

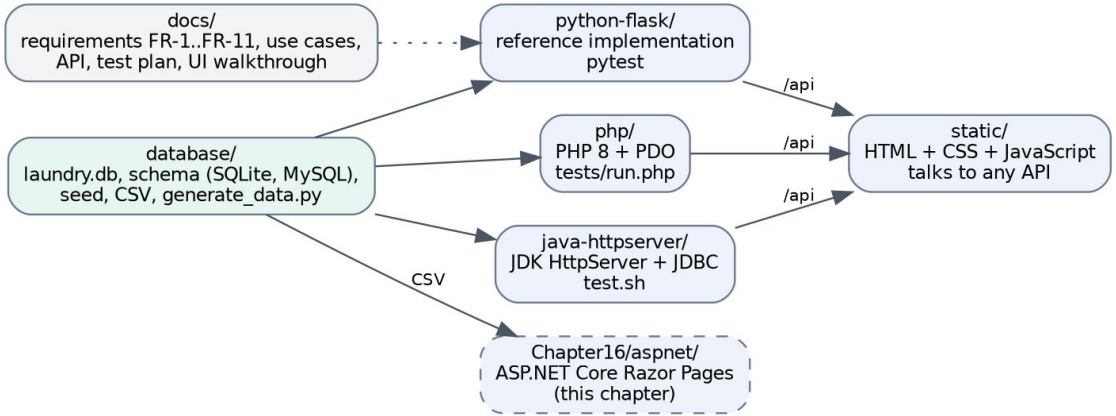


Figure 16.13 The complete Profitina Laundry application: one database, one set of documents, several tracks. The dashed box is the ASP.NET site of this chapter, which reads the same data as CSV.

Figure 16.13: one database, one set of documents (requirements FR-1 to FR-11, use cases, API, test plan), and tracks in Flask, PHP, Java, static HTML and this chapter's ASP.NET site. Book: Section 16.8.

More Than Four Hundred Checks

Flask (`python -m pytest`), **PHP** (`php tests/run.php`), **Java** (`./test.sh`) and the **static** front end (`node tests/run-tests.js`):

```
185 passed, 7 skipped in 52.25s
PASS 100   FAIL 0   MANUAL 3   SKIP 0   (total 103)
RESULT: 107 passed, 0 failed, 4 not automated/skipped  (17.4 s)
PASS 26   FAIL 0   (total 26)
```

Together with the 37 checks each of today's three sites passes; the sample database's SHA-256 was unchanged afterwards.

Read the application in the order of a request: once you can find the price computation, the permission check and the `INSERT` of a bill in each track, you can read any web application of this style. Book: Section 16.8.

The Problems Compared, 16.1–16.6

Table 16.4 The six problems of the project, and what each tempting solution gets wrong.

Problem	Topic	Main idea	What the tempting solution gets wrong
16.1	database design	named keys and relations, delete rule chosen on purpose	a default `DataRelation` cascades a deletion into fifty rows of history
16.2	authentication	hashed passwords, encrypted cookie, policies per folder	hidden links: an anonymous visitor opens `/customers`
16.3	CRUD	input models, the store behind an interface, Post/Redirect/Get	overposting: a cashier adds Rp 5.000.000 of deposit
16.4	design	one layout, one style sheet, measured contrast	light grey text at 2.32:1
16.5	evidence	an acceptance test over HTTP, scripted screenshots	screenshots alone prove nothing about refusals
16.6	deployment	publish, configure from outside, keep the key ring	keys lost at restart: everybody is logged out

Read down the last column of Table 16.4: the tempting solution works on the ordinary example and fails on one input nobody honest sends. Book: Section 16.9.

The Problems Compared, 16.7–16.13

Table 16.5 The seven new problems, and what each tempting solution gets wrong.

Problem	Topic	Main idea	What the tempting solution gets wrong
16.7	grid paging, sorting	clamp the page, parameters for values, a list for names	`Ma'ruf` breaks the filter; `ORDER BY` leaks a hash
16.8	grid footer	page total and month total, each labeled	a “month total” summed from one page
16.9	validation	attributes and `Validate`, `ModelState.IsValid` on the server	a bill of minus five kilograms
16.10	state	one right place per value	a `static` list shared by every user
16.11	reading data	one query or one relation, not one per row	$N+1$ round trips
16.12	transactions	several writes of one fact commit together	a bill without its log row
16.13	concurrency	optimistic check of the original values	last write wins: Dewi’s correction vanishes

In almost every row of Table 16.5 the failure is invisible from the browser of an honest user. Book: Section 16.9 and Table 16.7.

Where to Go Next

Frameworks: Django, Spring Boot, Laravel, ASP.NET Core MVC; find in each the pieces you already know — hashing, binding, validation, anti-forgery, transactions, sessions.

Deployment: HTTPS, a real database with tested backups, logs and a `/health` address, and a pipeline that deploys only if the tests pass.

Security: the OWASP Top 10 as a checklist; and UU PDP No. 27/2022 for personal data: collect only what the service needs, and delete it when there is no reason to keep it.

> **A READING PATH.** For the web itself: the HTTP specification, RFC 9110, read once from beginning to end. For security: the OWASP Top 10 and the OWASP Cheat Sheet Series. For data: Fowler's *Patterns of Enterprise Application Architecture*. Then the documentation of the framework you choose.

Choose a framework by language and ecosystem, not fashion. Book: Section 16.11 and Tables 16.9–16.10.

Profitina in Practice

Accounts and personal data: Argon2id hashes, rate-limited logins, consent, export, change of e-mail, deletion.

One database, two engines, PostgreSQL and SQLite, kept honest by a guard; **a dashboard in one request**, the lesson of Equation (16.10).

Deployment with a gate: a release counts only after its health check answers and more than a thousand tests pass.

Failures said out loud: when a model is unavailable, the application says so; a failed post is restored.

The product advises and its users decide; the runnable version of every idea is the Profitina Laundry code. Book: Section 16.12.

Common Mistakes

A default delete rule; hidden links instead of policies; binding more than the form may change; colors chosen by eye.

Evidence that cannot be re-run; deploying the laptop; the visitor's text in a query or an expression.

A footer summed from the screen; validating only in the browser; per-user data in a `static` field.

One question per row; two writes of one fact without a transaction; writing back a stale copy; a query value named `page`.

Each mistake points to the problem that shows its fix. Book: Section 16.13.

Summary: Five Things to Remember

1. A complete application is the eight steps of Chapter 8 plus three decisions (Table 16.1): Razor Pages, an interface for the data layer, one home per value.

2. Rules live on the server: prices by Equation (16.1), policies on folders, validation by `ModelState` (Problems 16.2, 16.3, 16.9).

3. Grids clamp and page by arithmetic (Equation (16.7)), take names from a list, and label every total (Equation (16.9)).

4. The data layer asks once (Equation (16.10)), writes one fact in one transaction, and saves only what is still as it was (Equation (16.11)).

5. Evidence is a test that anyone can re-run; a deployment keeps its data and keys, and secrets stay out of the repository.

Book: Section 16.14; Exercises 16.1–16.12 with solutions in Sections 16.15 and 16.16.

A Closing Word

The book began with a single request and ends with a shop: customers, bills, payments and people, on three outlets and in four languages, tested and deployed.

The technologies will change, as Web Forms gave way to Razor Pages; the questions will not.

Who may do this? Where does this value live? What if two people do it at once? What if the second write fails? How will I prove it works?

Ask those questions of every page you write, and you will write pages that people can trust with their laundry, and with much more. Book: Sections 16.10 and 16.14.